Tutorial: Hardware-Software Co-Design using Xilinx Vivado 2016.3 and the Xilinx SDK

Tutorial for the Nexys A7 FPGA Trainer Board

September 8, 2019

1 Introduction

The objective of this tutorial is to introduce Hardware/Software Co-Design using the Xilinx Vivado IDE and the Xilinx Software Development Kit (SDK).

In this tutorial you will learn the following topics:

1. How to design a hardware system in the Xilinx Vivado IP Integrator.
2. How to configure that system for the Digilent Nexys A7 Board using the Artix-7 FPGA.
3. How to create a software project for an implemented hardware design.
4. How to load a hardware system onto the Nexys A7 board and execute a software program.

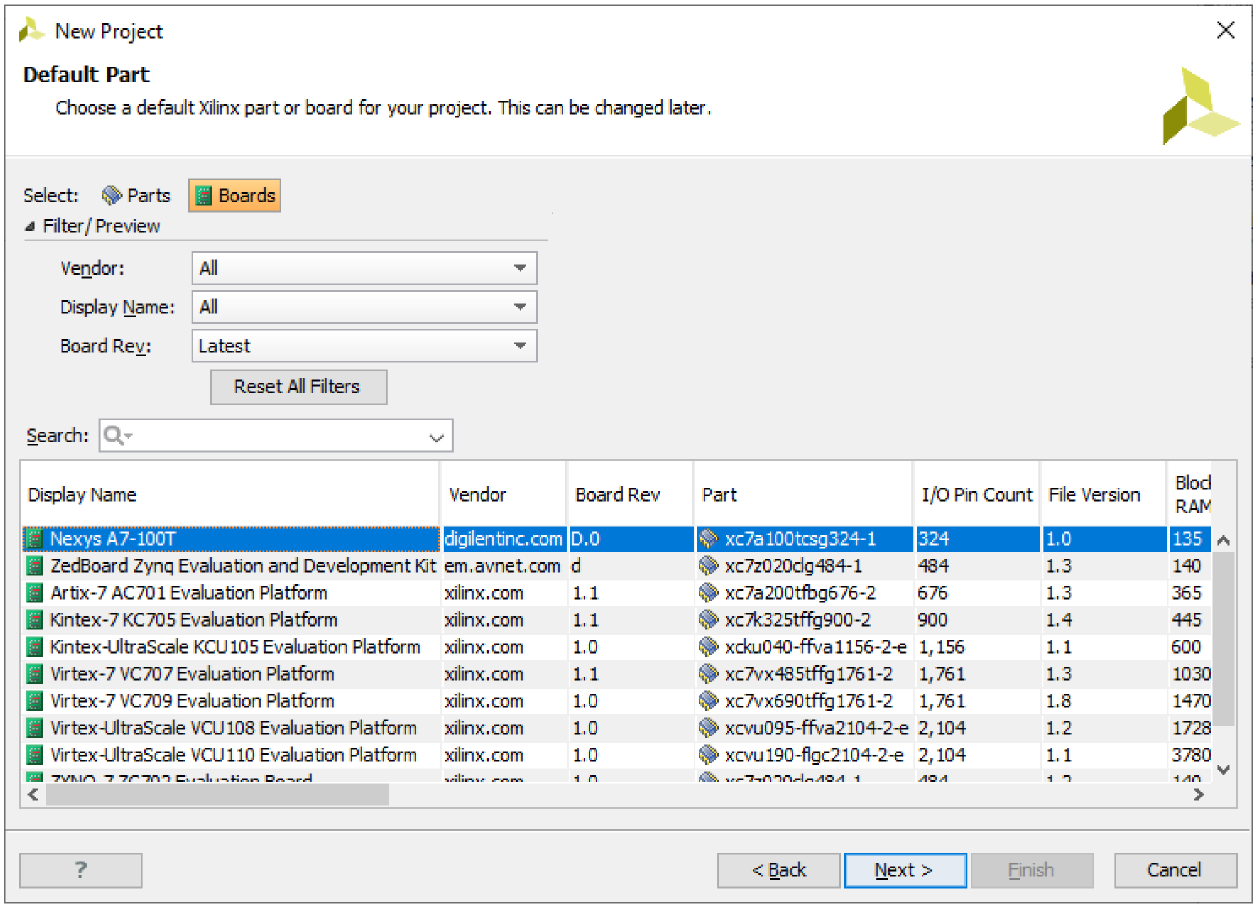
To complete this tutorial, you will need a Nexys A7 board, a micro USB cable, and access to a computer with the Xilinx Vivado IDE and Xilinx SDK software installed.

2 Preparing the Vivado IDE

In this exercise, we will create a new project in Vivado IDE by moving through the stages of the Vivado IDE *New Project Wizard*.

1. Start by launching the Vivado IDE: ⊞ → **Xilinx Design Tools** → **Vivado 2016.3**.
2. When Vivado loads, you will be presented with a splash screen. On the left side are several options; the right side allows you to choose from previous projects.
3. Select the option to Create New Project and the *New Project Wizard* will open. Click Next >.
4. Name your project **microblaze_tutorial** and set your project location to **H: engg3050**, Ensure that Create Project Subdirectory is selected, then click Next >.
5. At the *Project Type* dialogue, select RTL Project and select the option Do not specify sources at this time. Click Next >.

6. Next, select the board, using the window shown in figure 1. Select the Boards option at the top, then choose the Nexys A7-100T, board revision D.0, from the list. You can enter **Nexys** in the Search box to narrow your options.



Display Name	Vendor	Board Rev	Part	I/O Pin Count	File Version	Bloc RAM
Nexys A7-100T	digilentinc.com	D.0	xc7a100tcsg324-1	324	1.0	135
ZedBoard Zynq Evaluation and Development Kit	em.avnet.com	d	xc7z020clg484-1	484	1.3	140
Artix-7 AC701 Evaluation Platform	xilinx.com	1.1	xc7a200tfbg676-2	676	1.3	365
Kintex-7 KC705 Evaluation Platform	xilinx.com	1.1	xc7k325tffg900-2	900	1.4	445
Kintex-UltraScale KCU105 Evaluation Platform	xilinx.com	1.0	xcku040-ffva1156-2-e	1,156	1.1	600
Virtex-7 VC707 Evaluation Platform	xilinx.com	1.1	xc7vx485tffg1761-2	1,761	1.3	1030
Virtex-7 VC709 Evaluation Platform	xilinx.com	1.0	xc7vx690tffg1761-2	1,761	1.8	1470
Virtex-UltraScale VCU108 Evaluation Platform	xilinx.com	1.0	xcvu095-ffva2104-2-e	2,104	1.2	1728
Virtex-UltraScale VCU110 Evaluation Platform	xilinx.com	1.0	xcvu190-flgc2104-2-e	2,104	1.1	3780

Figure 1: Selecting the Nexys A7-100T board.

After selecting the correct board, click Next >.

7. A summary window will display your choices. If they are all correct, click Finish to create the project.

Now that we have created the project in Vivado IDE, we can now move on to creating a block design.

3 Creating a System Design

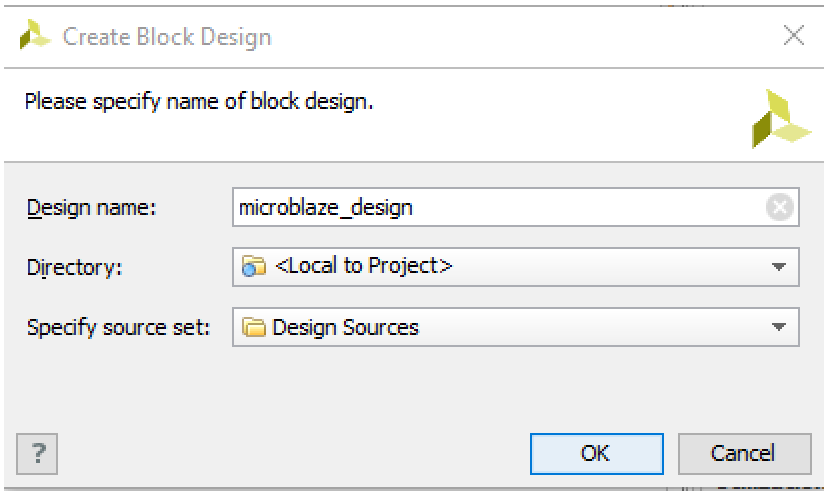
In this exercise, we will create an embedded microcontroller system and a General Purpose Input/Output (GPIO) controller within the general-purpose memory and logic fabric of the Artix-7

FPGA. The design uses a MicroBlaze soft microprocessor core and a GPIO controller. The GPIO controller is connected to the LEDs on the Nexys A7 board. It will also be connected to the MicroBlaze core via an AXI bus connection, allowing the LEDs to be controlled by a software application which we will create later in this tutorial.

We will begin by creating a new Block Design in the Vivado IDE.

1. Select the Create Block Design option in the *Flow Navigator* pane on the left. The *Create Block Design* dialog box will open.

2. Name your design **microblaze_design** as shown in Figure 2, and click OK. After a moment, a canvas will open, showing the diagram of your design.


Create Block Design

Please specify name of block design.

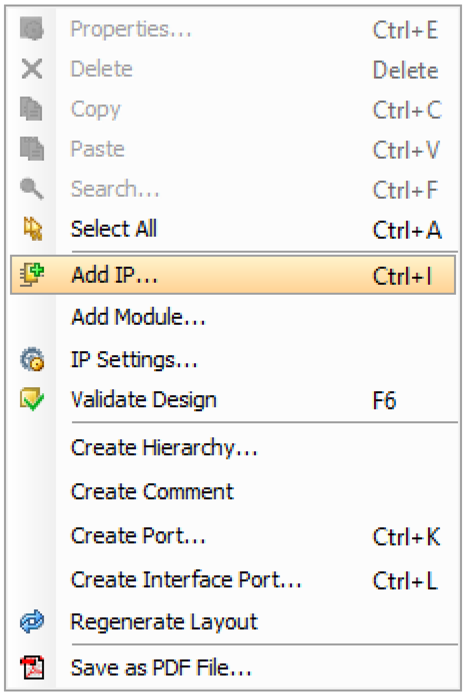
Design name: microblaze_design

Directory: <Local to Project>

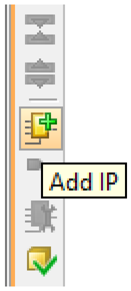
Specify source set: Design Sources

Figure 2: *Create Block Design* dialog box.

3. The first IP block that we will add to our design will be the MicroBlaze soft processor. In the *Vivado IP Integrator Diagram* canvas, right-click anywhere and select Add IP, as shown in Figure 3a. Alternatively, select the Add IP button in the toolbar at the left of the canvas, as shown in Figure 3b.



(a) Add IP option.



(b) Add IP from toolbar.

Figure 3: Two ways to add an IP block to your design.

The pop-up *IP Catalog Window* will open.

4. Enter **microblaze** into the search field and select the MicroBlaze core, as shown in Figure 4. Double-click the selection, or press Enter, to select the component.

   You should see a similar message to the following in the *Tcl Console* window to confirm that the processing system has been added to the design correctly:

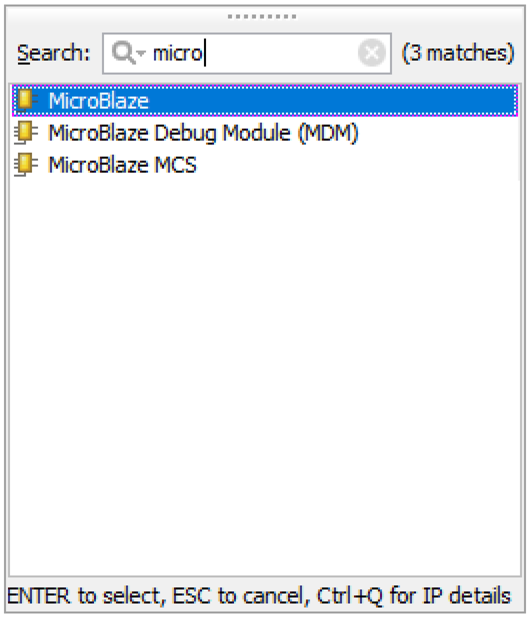
   ```
   create_bd_cell -type ip -vlnv xilinx.com:ip:microblaze:10.0 microblaaze_0
   ```

   Messages like this will be displayed in the *Tcl Console* window for all actions carried out on IP Integrator blocks.

5. The next step is to connect the interface ports on the MicroBlaze to the top-level interface ports on the design.

   Click the Run Block Automation option from the *Designer Assistance* message at the top of the Diagram window, as shown in Figure 5.

   The MicroBlaze is a highly-configurable soft processor core, with many options available to the designer. We will instantiate a simple microprocessor with an AXI port, as our design will not require interrupts nor see any performance improvements from cache memory, but will require the use of the AXI bus to interface with peripherals.



Search: micro (3 matches)

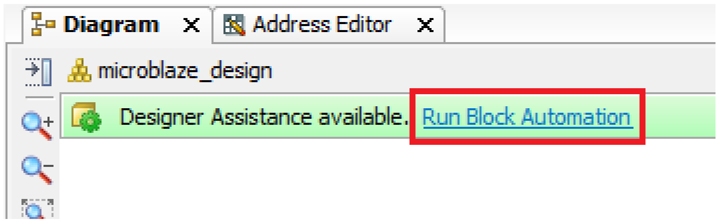
MicroBlaze

MicroBlaze Debug Module (MDM)

MicroBlaze MCS

ENTER to select, ESC to cancel, Ctrl+Q for IP details

Figure 4: *Add IP* selection list.



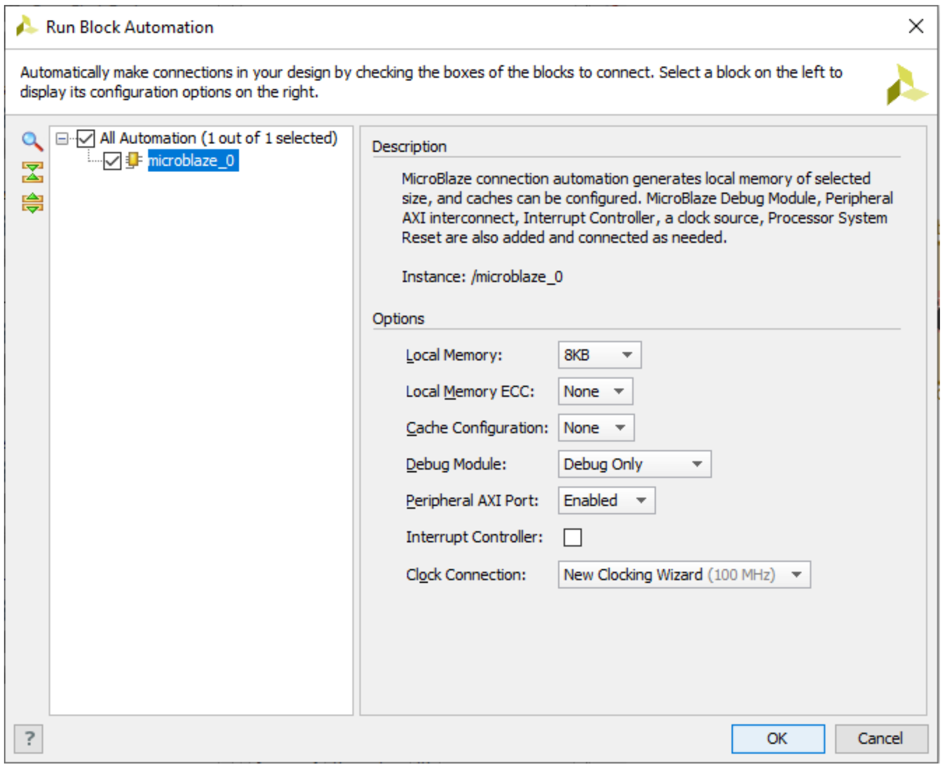
Designer Assistance available. Run Block Automation

Figure 5: Run Block Automation option.

Configure your MicroBlaze with the following settings, as shown in Figure 6:

- Local Memory: 32KB
- Local Memory ECC: None
- Cache Configuration: None
- Debut Module: Debug Only
- Peripheral AXI Port: Enabled
- Interrupt Controller: *unchecked*
- Clock Connection: New Clocking Wizard (100 MHz)

Click OK.



Automatically make connections in your design by checking the boxes of the blocks to connect. Select a block on the left to display its configuration options on the right.

Description

MicroBlaze connection automation generates local memory of selected size, and caches can be configured. MicroBlaze Debug Module, Peripheral AXI interconnect, Interrupt Controller, a clock source, Processor System Reset are also added and connected as needed.

Instance: /microblaze_0

Options

Local Memory: 8KB
Local Memory ECC: None
Cache Configuration: None
Debug Module: Debug Only
Peripheral AXI Port: Enabled
Interrupt Controller: ☐
Clock Connection: New Clocking Wizard (100 MHz)

Figure 6: Run Block Automation dialog box.

Aside: Caches vs. Block Ram

As you will remember from your computer architecture course, cache is meant to optimize performance when accessing memory that is not on-chip. However, since block RAM is on-chip inside the Artix-7, it is as fast as it can be, so there will be no performance improvement if cache memory is used.

Cache is only necessary if the memory requirements of your design cannot fit entirely within block RAM. Thus, if off-chip memory is required for your project, you will see performance improvements to your design when using cache. In this case, as a general rule, the cache should be sized to hold the data and the inner program loop of your program.

The Artix-7 is a powerful chip with 4.8 Mb (about 600 KB) of block RAM, implemented as 135 36-kb blocks — well above the minimal requirements of this lab.

The block diagram will be populated with supporting logic requierd for the MicroBlaze to operate: a MicroBlaze Debug Module to enable JTAG debugging in the software development process; a Processor System Reset block to interface the on-board reset button with functions to clear the memory and restart the processor; the local memory module which implements the RAM; and the Clocking Wizard which uses on-chip PLLs to generate multiple system clocks with selectable frequency and phase as needed for the design.

Note that any IP module can be double-clicked to "look inside" and change settings or see the details of sub-modules. First, however, we should finish adding the basic elements of our design by adding the GPIO controller.

6. Click on the Add IP button, and add an **AXI GPIO** block to the design. Search for `GPIO` in the search bar to narrow down your choices, then double-click on AXI GPIO to add the block.

Aside: The AXI Standard

AXI stands for Advanced eXtensible Interface, and the current version is AXI4, which is part of the ARM AMBA 3.0 open standard. Many devices and IP blocks produced by third party manufacturers and developers are based on this standard.

AXI buses can be used flexibly, and in the general sense are used to connect the processor(s) and other IP blocks in an embedded system. In fact there are three flavours of AXI4, each of which represents a different bus protocol, as summarised below. The choice of AXI bus protocol for a particular connection depends on the desired properties of that connection.

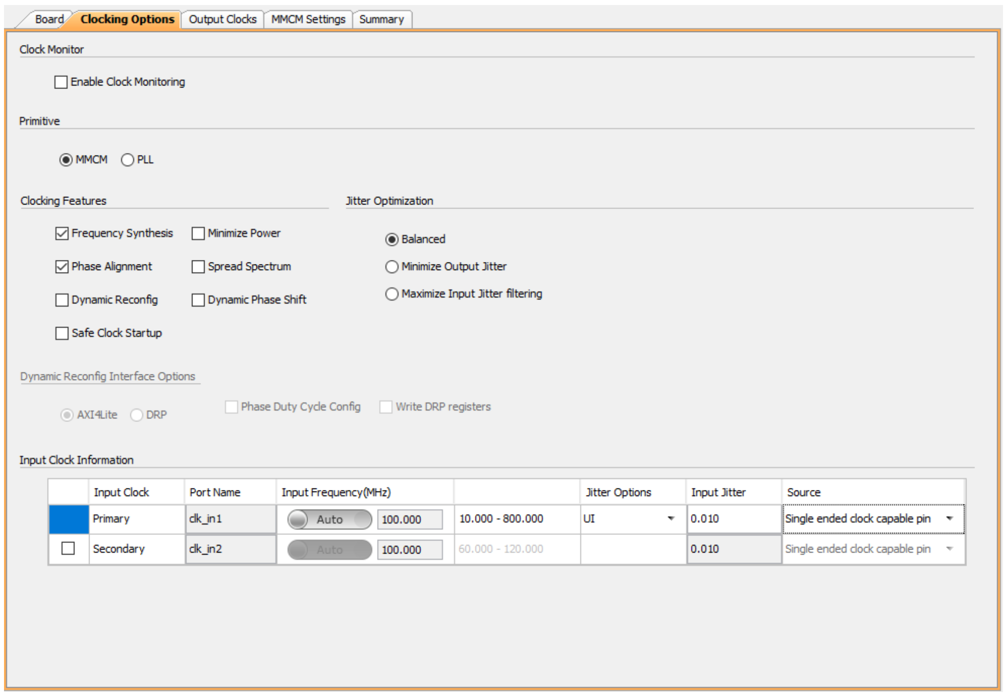
- AXI4 — For memory-mapped links, and providing the highest performance: an address is supplied followed by a data burst transfer of up to 256 data words (or data beats).
- AXI4-Lite [2] A simplified link supporting only one data transfer per connection (no bursts). AXI4-Lite is also memory-mapped: in this case an address and single data word are transferred.
- AXI4-Stream [1] For high-speed streaming data, supporting burst transfers of unrestricted size. There is no address mechanism; this bus type is best suited to direct data flow between source and destination (non memory mapped).

The term memory mapped is used in the above descriptions, and it is useful to briefly confirm its meaning. If a protocol is memory mapped, an address is specified within the transaction issued by the master (read or write), which corresponds to an address in the system memory space. In the case of AXI4-Lite, which supports a single data transfer per transaction, data is then written to, or read from, the specified address; in the case of AXI4 bursts, the address specified is for the first data word to be transferred, and the slave must then calculate the addresses for the data words that follow.

For more details on the AXI interface and interconnects, please consult Chapter 2 of *The Zynq Book*, www.zynqbook.com.

7. Before you connect everything, there are a few settings that must be adjusted. By default, the Clocking Wizard expects a differential clock input and an active-high reset switch, but the Nexys A7 has a single clock crystal signal and an active-low reset.

8. Double-click on the Clocking Wizard to open its settings.


Board | **Clocking Options** | Output Clocks | MMCM Settings | Summary

Clock Monitor

☐ Enable Clock Monitoring

Primitive

[✓ MMCM] PLL

Clocking Features

☑ Frequency Synthesis ☐ Minimize Power
☑ Phase Alignment ☐ Spread Spectrum
☐ Dynamic Reconfig ☐ Dynamic Phase Shift
☐ Safe Clock Startup

Jitter Optimization

[✓ Balanced]
Minimize Output Jitter
Maximize Input Jitter filtering

Dynamic Reconfig Interface Options

[✓ AXI4Lite] DRP ☐ Phase Duty Cycle Config ☐ Write DRP registers

Input Clock Information

	Input Clock	Port Name	Input Frequency(MHz)			Jitter Options	Input Jitter	Source
☑	Primary	clk_in1	Auto	100.000	10.000 - 800.000	UI	0.010	Single ended clock capable pin
☐	Secondary	clk_in2	Auto	100.000	60.000 - 120.000		0.010	Single ended clock capable pin

Figure 7: The Clocking Wizard settings dialog box.

9. In the Clocking Options tab, under *Input Clock Information*, set the primary clock to **Single-ended clock capable pin**, as shown in Figure 7.

10. In the Output Clocks tab, under *Reset Type*, select Active Low. This tab is also where additional clocks are instantiated, if necessary. The designer can select the frequency, phase shift, and duty cycle for each clock.

    Note also that the block representing the Clocking Wizard changes depending on the settings selected by the designer. It should look like Figure 8. Click OK.

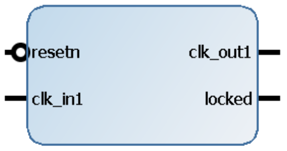


Figure 8: Clocking Wizard block with active-low reset and single-ended clock input.

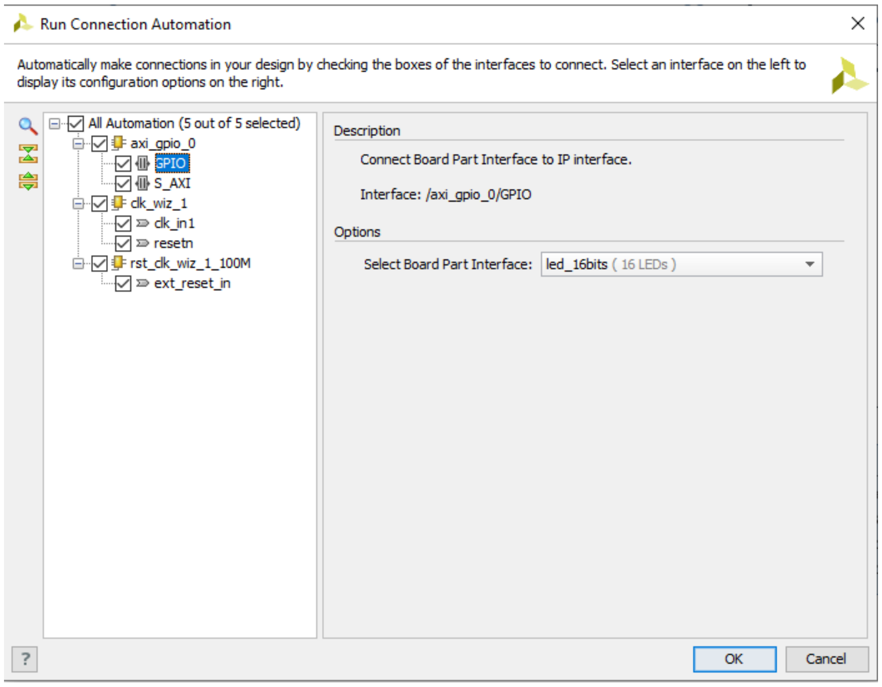
    Now, you can connect the IP blocks to each other and to the external components on the Nexys A7 board.

11. Select Run Connection Automation in the green bar at the top of the screen. The *Run Connection Automation* dialog box will open.

In this box, you can set the parameters for connections for three components: the GPIO module, the Clock Wizard, and the Reset module.

12. Under axi_gpio_0, select **GPIO**. As you are interfacing with the 16 on-board LEDs: under *Select Board Part Interface*, choose led_16bits (16 LEDs). Note the other options here: the AXI GPIO block can also be used to drive the switches, pushbuttons, seven-segment displays, and RGB LEDs.

13. Next, select the *All Automation* check box to enable automation for all five available signals, as shown in Figure 9. Click OK.



Automatically make connections in your design by checking the boxes of the interfaces to connect. Select an interface on the left to display its configuration options on the right.

Connect Board Part Interface to IP interface.

Interface: /axi_gpio_0/GPIO

Select Board Part Interface: led_16bits (16 LEDs)

Figure 9: Run Connection Automation dialog box.

The input ports sys_clock and reset, and the output port led_16bits, along with seven IP blocks, should now be visible on your design. Your design should be similar to Figure 10, but the positions of ports, nets and blocks may vary slightly.

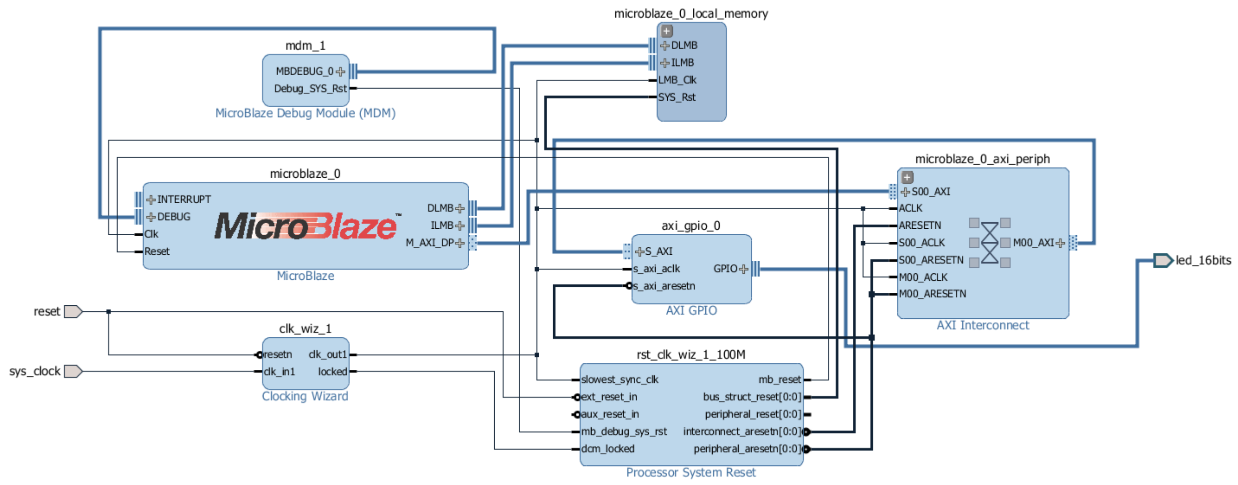
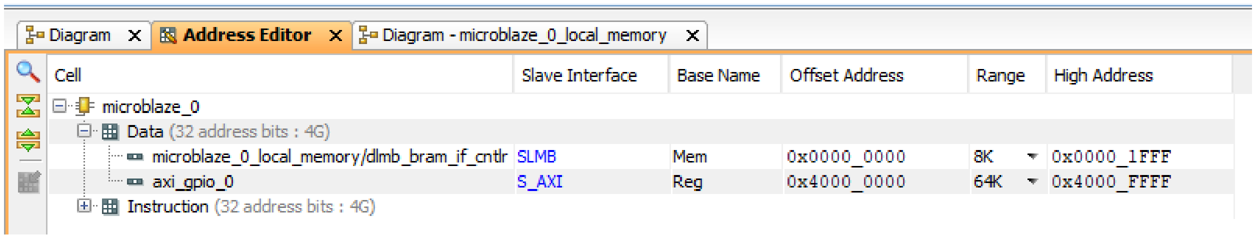


Figure 10: Completed MicroBlaze GPIO design.

IP Integrator will automatically assign a memory map for all IP that is present in the design. We will not be changing the memory map in this tutorial, but for future reference we will take a look at the Address Editor.

14. Select the Address Editor tab from the top of the *Workspace* window, as shown in Figure 11, and expand the *Data* group.



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
microblaze_0					
Data (32 address bits : 4G)					
microblaze_0_local_memory/dlmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	8K	0x0000_1FFF
axi_gpio_0	S_AXI	Reg	0x4000_0000	64K	0x4000_FFFF
Instruction (32 address bits : 4G)					

Figure 11: Address Editor tab.

You can see that IP Integrator has already assigned a memory map (the mapping of specific sections of memory to the memory-mapped registers of the IP blocks) to the AXI GPIO interface, and that it has a range of 64K. A different block of addresses has been assigned to the instruction and data memory for the MicroBlaze core.

Now that our system is complete, we must first validate the design before generating the HDL design files.

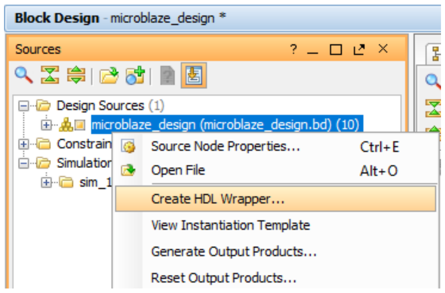
15. Save your design by selecting **File** → **Save Block Design** from the *Menu Bar*.

16. Validate the design by selecting **Tools** → **Validate Design** from the *Menu Bar*. This will run a Design Rule Check (DRC). After a moment, *Validate Design* dialogue should appear to confirm that validation of the design was successful. Click OK to dismiss the message.

With the design successfully validated, we can now move on to generating the HDL design files for the system.

17. Switch to the *Sources* tab by selecting **Window** → **Sources** from the menu bar.

18. In the *Sources* window, right-click on the top-level system design, which in this case is **microblaze_design**, and select Create HDL Wrapper, as shown in Figure 12.



Block Design - microblaze_design *
Sources
Design Sources (1)
microblaze_design (microblaze_design.bd) (10)
Constrain
Simulation
sim_1
Source Node Properties... Ctrl+E
Open File Alt+O
Create HDL Wrapper...
View Instantiation Template
Generate Output Products...
Reset Output Products...

Figure 12: Address Editor tab.

The *Create HDL Wrapper* dialog will open. Select Let Vivado manage wrapper and auto-update and click OK. This will generate the top-level HDL wrapper for our system.

All of the source files for the IP blocks that were used in the IP Integrator block diagram, as well as any relevant constraints files, will be generated during the generation of the HDL wrapper. As the target language was specified as VHDL, all generated source files are VHDL.

With all HDL design files generated, the next step in Vivado is to implement our design and generate a bitstream file.

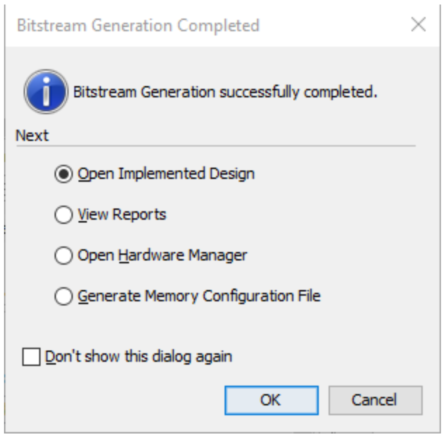
19. In the *Flow Navigator* pane, select Generate Bitstream from the *Program and Debug* section. If a dialog window appears prompting you to save your design, click Save. If Vivado prompts you to launch synthesis and implementation first, click Yes to accept.

    The combination of running synthesis, implementation and bistream generation may take up to ten minutes. Progress can be monitored using the progress bar in the top right corner of the Vivado window, and messages will appear in the tabs of the *Results Window Area* pane at the bottom of the screen.

20. Once the bitstream has been generated, a dialog window will open to inform you that the process has been completed successfully, as shown in Figure 13.

21. Select Open Implemented Design and click OK. At this point you will be presented with the *Device* view, where you can see the resources utilized by the design. With the default colour scheme, these are shown in light blue.

    With the bitstream generated, the building of the hardware image is complete. It must now be exported to a software environment where we will build a software application to control and interact with the custom hardware.


Bitstream Generation Completed

Bitstream Generation successfully completed.

Next

● Open Implemented Design

○ View Reports

○ Open Hardware Manager

○ Generate Memory Configuration File

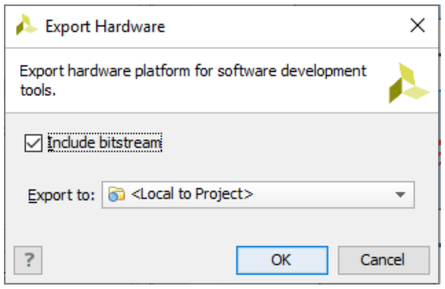
☐ Don't show this dialog again

Figure 13: Bitstream generation completed.

The final step in Vivado is to export the design to the SDK, where we will create the software application that will allow the MicroBlaze processor to control the LEDs on the Nexys A7 board.

22. Select **File** → **Export** → **Export Hardware...** from the *Menu Bar*.

23. The *Export Hardware* dialog window will open. Ensure that the option to Include Bitstream is selected, as in Figure 14, then click OK.


Export Hardware

Export hardware platform for software development tools.

☑ Include bitstream

Export to: <Local to Project>

Figure 14: Export hardware for SDK.

NOTE: For the option to Include Bitstream to be eneabled, an implemented design must be active. This is the reason we opened the implemented design after bitstream generation in Step 20.

24. Launch the SDK in Vivado by selecting **File** → **Launch SDK** from the *Menu Bar* and click OK.

4 Creating a Software Application in the SDK

Next, we will create a simple software application which will control the LEDs on the Nexys A7 board. The software application wlil run on the MicroBlaze processor and communicate with the AXI GPIO block which is connected to the LEDs. We will take a look at the software drivers that are created by IP Integrator for each of the IP modules, before building and executing the software on the development board.

The SDK should have opened after the conclusion of the previous section. If it did not open, you can open the SDK by navigating to ⊞ → **Xilinx Design Tools** → **Xilinx SDK 2016.3** When launching the SDK from the start menu, you will need to specify the workspace that was created when the Vivado IP Integrator design was exported. It should be:

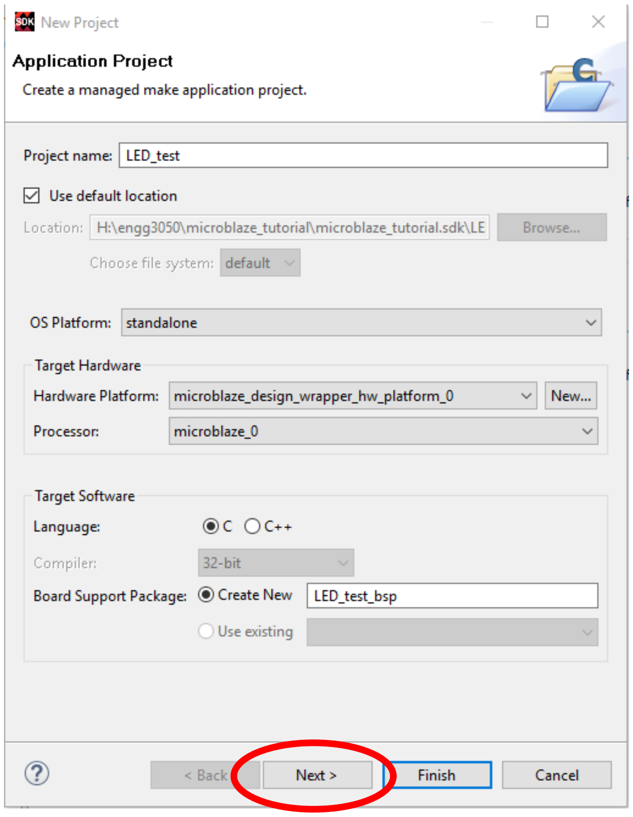
`H:\engg3050\microblaze_tutorial\microblaze_tutorial.sdk`

Enter this into the Workspace field of the *Workspace Launcher* dialog window when the SDK opens.

With the SDK open, we can begin the creation of our software application. You will already be able to see the Hardware Platform Project, which will automatically be created and opened. It is now necessary to add an Application Project and a Board Support Package.

- Select **File** → **New** → **Application Project** from the *Menu Bar*.
- The *New Project* dialog window will open. In the Project Name field, enter **LED_test**. Leave the other options as default, as shown in Figure 15. Click Next >. Do not click Finish.
- At the *New Project Templates* dialog box, select Empty Application, and click Finish.

  The project should open automatically. We can now import some pre-made code for the application. The source code for this lab is contained in the file `LED_test_microblaze.c` and can be found on the course website, or in the appendix of this lab.


New Project

Application Project

Create a managed make application project.

Project name: LED_test

☑ Use default location

Location: H:\engg3050\microblaze_tutorial\microblaze_tutorial.sdk\LE

Choose file system: default

OS Platform: standalone

Target Hardware

Hardware Platform: microblaze_design_wrapper_hw_platform_0

Processor: microblaze_0

Target Software

Language: [✓ C] C++

Compiler: 32-bit

Board Support Package: [✓ Create New] LED_test_bsp

Use existing

Figure 15: Creating a new Application Project in SDK.

- In the *Project Explorer* panel, expand newly-created project **LED_test** and highlight the *src* directory. Right-click and select Import... as shown in Figure 16.

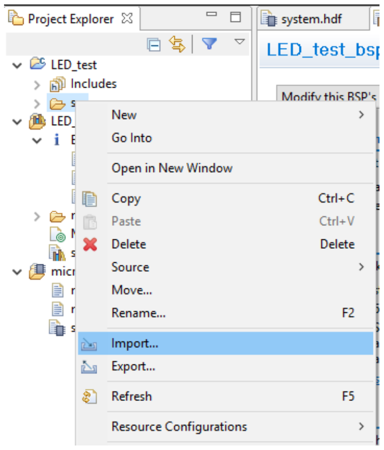
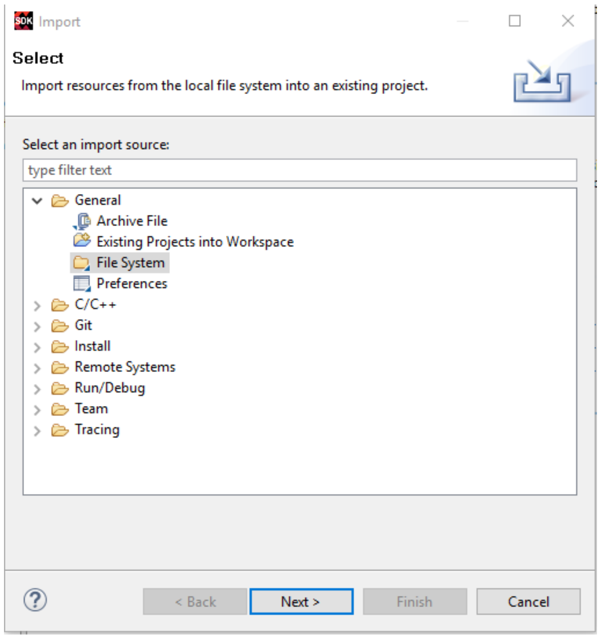
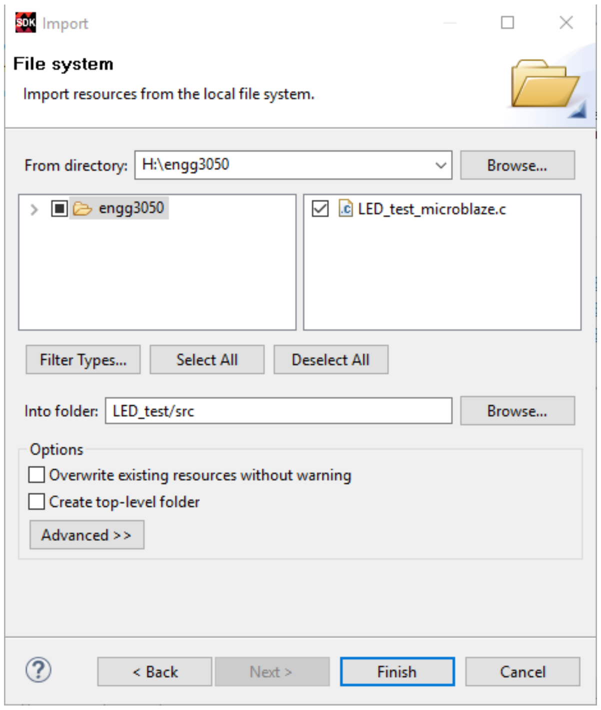


Figure 16: Import source files into project.

- The *Import* window will open. Expand the General option and highlight *File System*, as in Figure 17a, and click Next >.
- In the *Import File System* window, click the Browse... button. Browse to the directory location where you saved the source code file, and click OK.
- Select the file **LED_test_microblaze.c**, as shown in Figure 17b, and click Finish.



(a) Import dialog box.



(b) Import file system dialog.

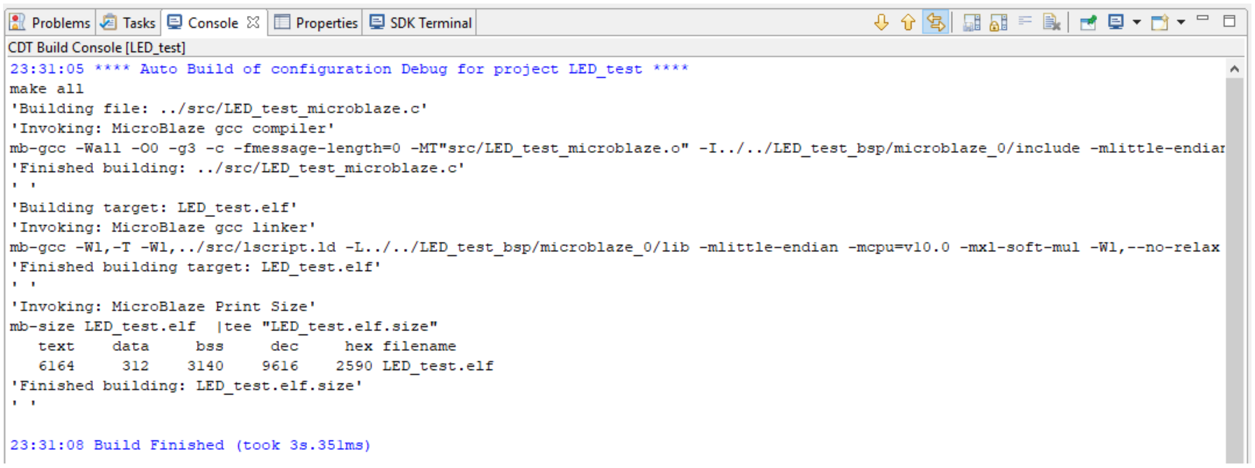
Figure 17: Importing code from a file within the file system.

The C source file will be imported and the project should automatically build. You should see a similar message to Figure 18 in the *Console* window.

- Expanding the *src* folder and double-click on **LED_test_microblaze.c** to open the source file and explore the code.

  Note the function `XGpio_Initialize(&Gpio, GPIO_DEVICE_ID)` which is a function provided by the GPIO device driver in the file `xgpio.h`. It initializes the XGpio instance *Gpio*, with the unique ID of the device specified by `GPIO_DEVICE_ID`.

  At the top of `LED_test_microblaze.c`, you will see that `GPIO_DEVICE_ID` is defined as `XPAR_AXI_GPIO_0_DEVICE_ID`. The value of this constant can be found by right-clicking its name in the code window, and clicking Open Declaration. This will open the file `xparameters.h`, an auto-generated file containing the definitions of all of the hardware parameters in the system.



```
23:31:05 **** Auto Build of configuration Debug for project LED_test ****
make all
'Building file: ../src/LED_test_microblaze.c'
'Invoking: MicroBlaze gcc compiler'
mb-gcc -Wall -O0 -g3 -c -fmessage-length=0 -MT"src/LED_test_microblaze.o" -I../../LED_test_bsp/microblaze_0/include -mlittle-endiar
'Finished building: ../src/LED_test_microblaze.c'
' '
'Building target: LED_test.elf'
'Invoking: MicroBlaze gcc linker'
mb-gcc -Wl,-T -Wl,../src/lscript.ld -L../../LED_test_bsp/microblaze_0/lib -mlittle-endian -mcpu=v10.0 -mxl-soft-mul -Wl,--no-relax
'Finished building target: LED_test.elf'
' '
'Invoking: MicroBlaze Print Size'
mb-size LED_test.elf  |tee "LED_test.elf.size"
   text    data     bss     dec     hex filename
   6164     312    3140    9616    2590 LED_test.elf
'Finished building: LED_test.elf.size'
' '

23:31:08 Build Finished (took 3s.351ms)
```

Figure 18: Building the project.

The function `XGpio_SetDataDirection(&Gpio, LED_CHANNEL, 0x0000)` is also provided by the GPIO device driver, and sets the direction of the specified GPIO port. As we are specifying LEDs, we want to set this port as an output. Bits set to '0' are outputs, and bits set to '1' are inputs. As there are 16 LEDs on the Nexys A7 board, by setting the direction to a value of `0x0000`, we are setting all 16 LEDs as outputs.

Further information on the peripheral drivers can be found by selecting the *system.mss* tab. A list of all peripherals in the system is provided, along with links to available documentation and examples.

The next step is to program the MicroBlaze with the bitstream file that we generated in Vivado.

- First, the three board jumpers must be set correctly. Set the power jumper (located beside the USB port) to USB, as shown in Figure 19.

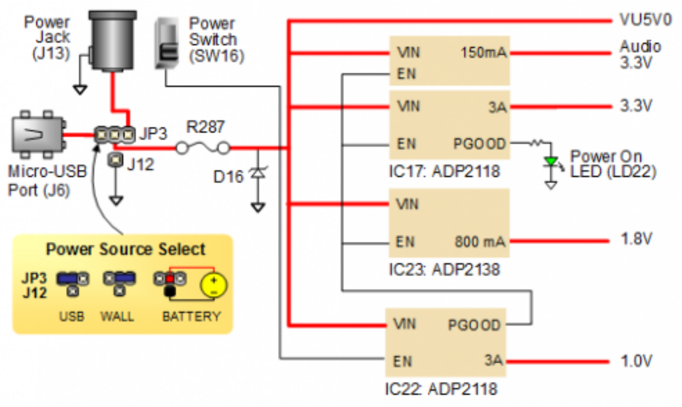


Figure 19: Settings for the power jumper.

- There are two programming jumpers: JP1, located near the VGA port, and JP2, located approximately one inch (3 cm) below the power switch. Set both JP1 and JP2 to USB, as shown in Figure 20.

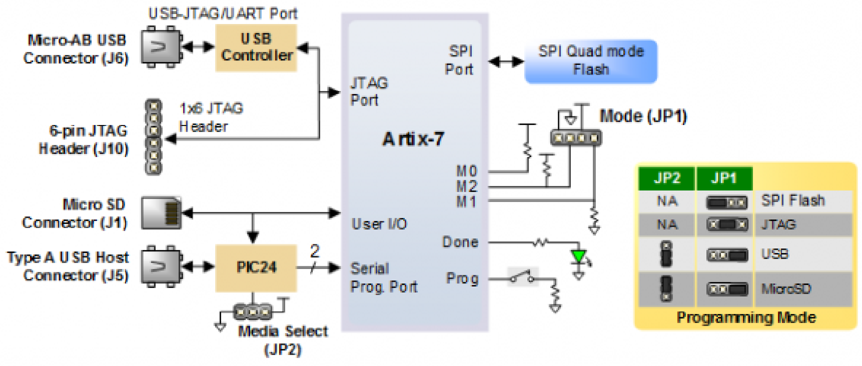
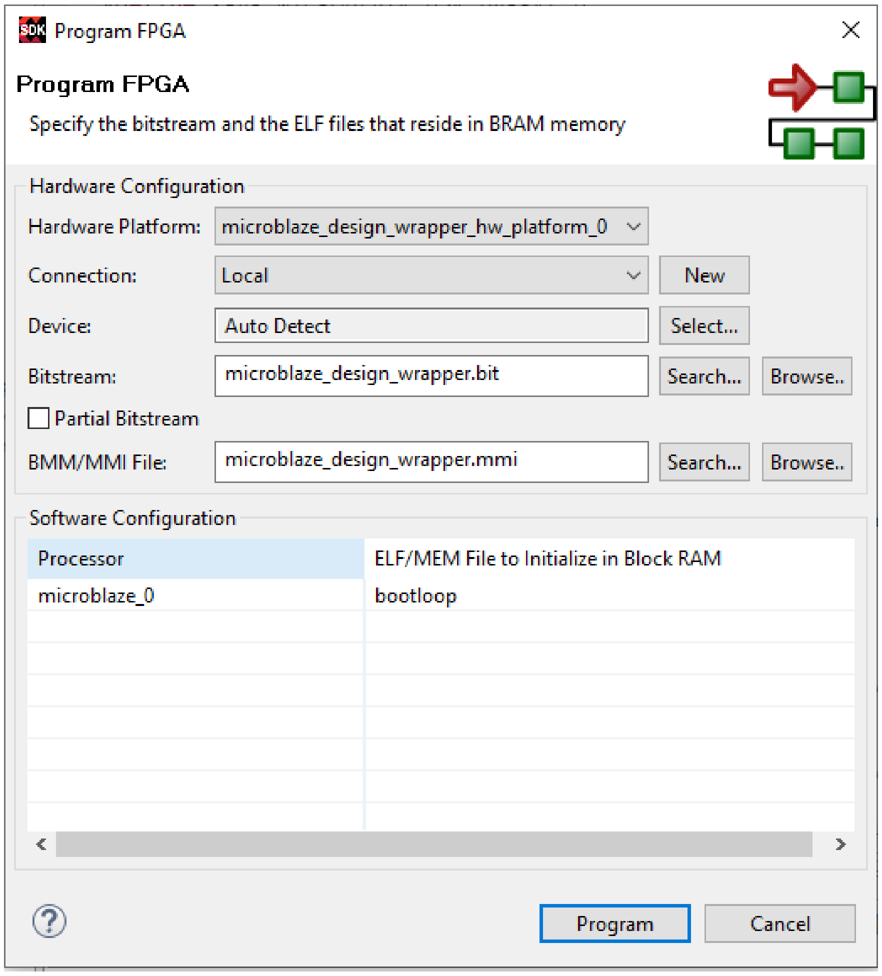


Figure 20: Settings for programming jumpers.

- Connect the **PROG UART** port of the Nexys A7 board to the PC via the provided USB cable, and power on the board.

- Download the bitstream by selecting Xilinx Tools → Program FPGA from the *Menu Bar* in the SDK. The *Program FPGA* window will appear. The *Bitstream* field should already be populated with the correct bitstream file, as shown in Figure 21.


Program FPGA

Program FPGA

Specify the bitstream and the ELF files that reside in BRAM memory

Hardware Configuration

Hardware Platform:	microblaze_design_wrapper_hw_platform_0		
Connection:	Local	New	
Device:	Auto Detect	Select...	
Bitstream:	microblaze_design_wrapper.bit	Search...	Browse..
☐ Partial Bitstream			
BMM/MMI File:	microblaze_design_wrapper.mmi	Search...	Browse..

Software Configuration

Processor	ELF/MEM File to Initialize in Block RAM
microblaze_0	bootloop

Figure 21: Building the project.

- Click Program. Once the device has been successfully programmed, the *DONE* LED on the Nexys A7 board will turn green.

  With the bistream programmed, we can now launch our software program on the Nexys A7.

- Select the top-level project **LED_test** in the *Project Explorer*. Right-click and select **Run As → Launch on Hardware (GDB)**

  After a few seconds, the LEDs on the Nexys A7 board should begin to flash between two states as shown in Figure 22.

In summary, a GPIO controller and MicroBlaze processor have been successfully implemented in the FPGA fabric of the Artix-7 device. The MicroBlaze processor was then programmed to control the LEDs by means of a standalone software application with the capability to interface with the GPIO controller in the FPGA fabric.

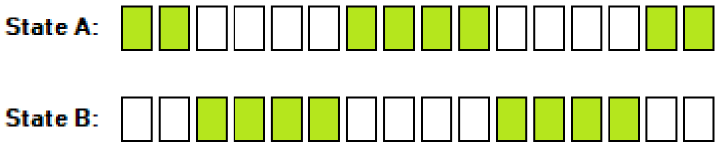


Figure 22: The two possible states of the flashing green LEDs.

5 Acknowledgements

This tutorial has been written and compiled by Matt Saunders. Images are of the Xilinx Vivado 2016.3 FPGA design suite including the Xilinx Software Development Kit. This tutorial is a derivative of the following work:

L.H. Crockett, R. A. Elliot, M. A. Enderwitz and D. Stewart, *The Zynq Book Tutorials for Zybo and Zedboard*, First Edition, Strathclyde Academic Media, 2015.

6 Appendix: LED_test_microblaze.c

```
/*
 * LED_test_microblaze.c
 * Created on:      14 July 2019
 *     Author:      Matt Saunders
 *
 * Adapted from LED_test.c by Ross Elliot, The Zynq Book Tutorials.
 */


/* Include Files */
#include "xparameters.h"
#include "xgpio.h"
#include "xstatus.h"
#include "xil_printf.h"

/* Definitions */
#define GPIO_DEVICE_ID  XPAR_AXI_GPIO_0_DEVICE_ID /* GPIO device */
#define LED 0xC3C3                  /* Initial LED value - XX0000XX_XX0000XX */
#define LED_DELAY 10000000          /* Software delay length */
#define LED_CHANNEL 1               /* GPIO port for LEDs */
#define printf xil_printf           /* smaller, optimised printf */

XGpio Gpio;                         /* GPIO Device driver instance */

int LEDOutputExample(void) {
    volatile int Delay;
    int Status;
    int led = LED; /* Hold current LED value. Initialise to LED definition */

        /* GPIO driver initialisation */
        Status = XGpio_Initialize(&Gpio, GPIO_DEVICE_ID);
        if (Status != XST_SUCCESS) {
            return XST_FAILURE;
        }

        /*Set the direction for the LEDs to output. */
        XGpio_SetDataDirection(&Gpio, LED_CHANNEL, 0x0000);

        /* Loop forever blinking the LED. */
            while (1) {
                /* Write output to the LEDs. */
                XGpio_DiscreteWrite(&Gpio, LED_CHANNEL, led);

                /* Flip LEDs. */
                led = ~led;

                /* Wait a small amount of time so the LED blinking is visible. */
                for (Delay = 0; Delay < LED_DELAY; Delay++);
            }

        return XST_SUCCESS; /* Should be unreachable */
}

int main(void) {
    int Status;

    /* Execute the LED output. */
    Status = LEDOutputExample();
    if (Status != XST_SUCCESS) {
        xil_printf("GPIO output to the LEDs failed!\r\n");
    }
    return 0;
}
```