ECE385
DIGITAL SYSTEMS LABORATORY
Introduction to MicroBlaze and Vitis SDK

Abstract and Goals

The goal of this lab is to create a MicroBlaze based system on the Spartan-7 FPGA. MicroBlaze is a soft-IP based 32-bit CPU which can be programmed using a high-level language (in this class, we will be using C). A typical use case scenario is to have the MicroBlaze be the system controller and handle tasks which do not need to be high performance (for example, user interface, data input and output) while an accelerator peripheral in the FPGA logic (designed using SystemVerilog) handles the high-performance operations. This is the idea behind the SoC, or System-on-Chip concept.

The following will give you a walkthrough of the block design editor tool which is used to instantiate IP blocks (including the MicroBlaze core). We will set up a minimal MicroBlaze system using on-chip memory blocks and a PIO (Parallel I/O) block to blink some LEDs using a C program running on the CPU core to confirm it is working. You will then be asked to write a program which reads 8-bit numbers from the switches on the FPGA board and sums into an accumulator, displaying the output using the LEDs via the MicroBlaze. This will involve instantiating another PIO block to read data from the switches and modifying the C program to input data, add, and display the data.

Be prepared to give answers to any of the italicized questions in this document from your TA when demoing. This is to ensure that you try to research what the settings do instead of simply trying to "make the picture look like your screen".

Hints:
Unit-test the inputs and outputs. The outputs (LEDs) should already work, but make sure you can turn on and off every bit. If you have problems, check the schematic for the FPGA board, and make sure you are toggling the correct pins.

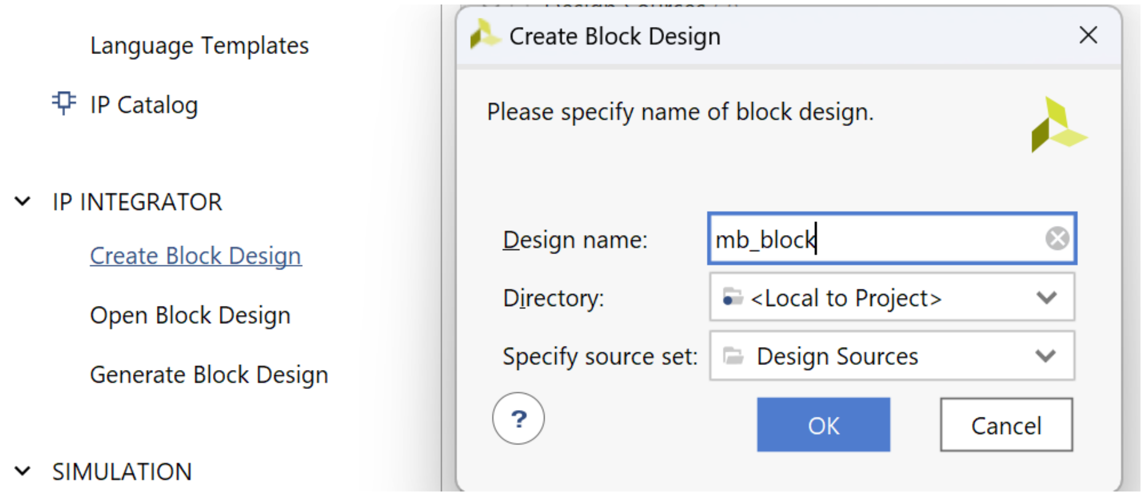
For this, and the rest of the class, you may use the C standard libraries (stdlib.h) or the C++ equivalents, this can save you a lot of work when coding in C. Note that to save memory, certain functions may be unavailable or be modified (e.g., printf may have certain limitations for floating point types). In addition, certain system functions such as the time and date (time.h) may be unavailable because your hardware platform has not been designed with a timer.

Set up the System Combining the FPGA with the MicroBlaze Processor:

Create a New Project:

- Follow the directions from the "Introduction to Xilinx Vivado" document to create a new (empty) project. Make sure you select the correct FPGA device.

- Make sure you create the project with a relatively short path without spaces. E.g., C:\Users\<netid>\ece_lab6\. Specifically, if you are on Windows, your path should not include any spaces. **Do not work directly from the U:\ drive on EWS machines, some parts of the Vitis build tools do not recognize network paths, so you will get seemingly arbitrary errors.**
- In your blank project, go to the "IP Integrator" category and click on "Create Block Design". Then name your block design (for our purposes, we have named this mb_block), but you might need to name this something else for future labs which use MicroBlaze. (See Figure 1)



Please specify name of block design.

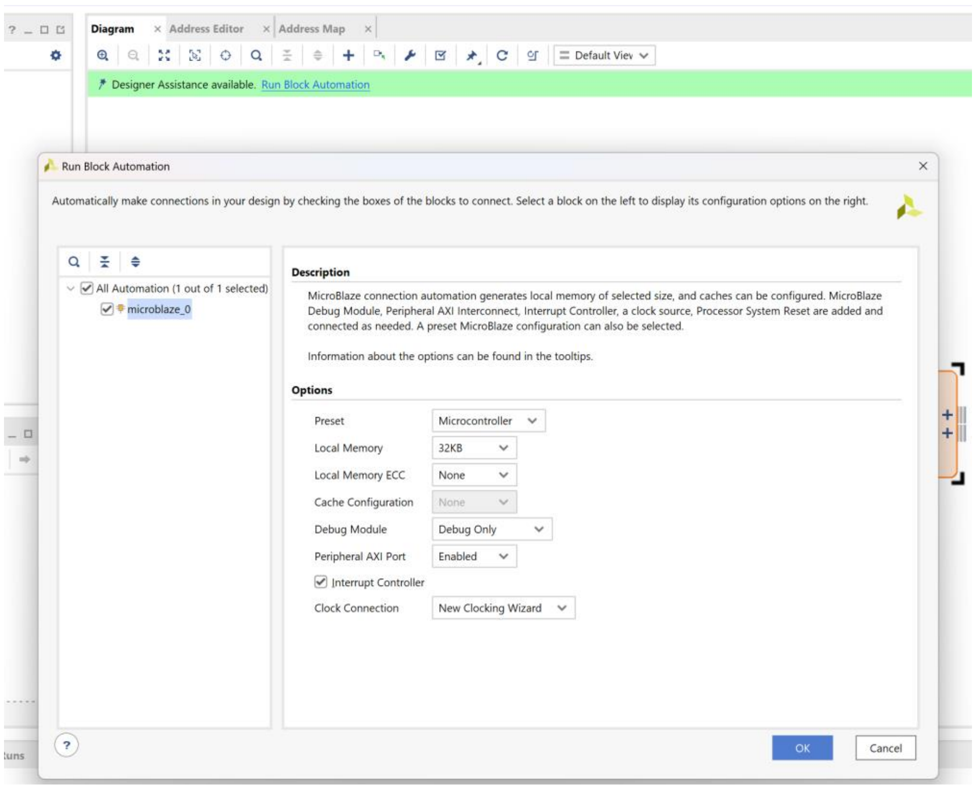
Design name: mb_block

Directory: <Local to Project>

Specify source set: Design Sources

Figure 1 – Creating the Block Design

- You will be greeted with a blank block design – upon which you can hit the (+) symbol to start adding IP blocks. The first IP block you should add is "MicroBlaze", which will then add an unconfigured MicroBlaze processor to your diagram.
- The MicroBlaze processor core is not too useful by itself. You will need to populate your SoC design with memories, peripherals, etc. to make for a useful design. The bus architecture for connecting peripherals to the MicroBlaze processor is called the AXI bus (Advanced eXtensible Interface). This interface was developed by ARM Ltd. is the same interface that ARM uses for connecting ARM processors to peripherals (though the MicroBlaze processor is not an ARM-based core). Fortunately, Vivado provides some presets for commonly used configurations. You can load these templates by clicking on "Run Block Automation".
- Select the "Microcontroller" Preset and then modify the "Local Memory" to 32KB (see Figure 2). *You should do some research and figure out what are some primary differences between the various presets which are available.*



Designer Assistance available. Run Block Automation

Run Block Automation

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.

☑ All Automation (1 out of 1 selected)
☑ microblaze_0

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 added and connected as needed. A preset MicroBlaze configuration can also be selected.

Information about the options can be found in the tooltips.

Options

Preset	Microcontroller
Local Memory	32KB
Local Memory ECC	None
Cache Configuration	None
Debug Module	Debug Only
Peripheral AXI Port	Enabled
☑ Interrupt Controller	
Clock Connection	New Clocking Wizard

OK Cancel

Figure 2 - Block Automation

Pressing OK will create a design as shown in Figure 3. Note that it's clear what some of the components are (e.g., the MicroBlaze processor), while others are less obvious. Make sure you do a little bit of research regarding each component and can summarize their functionality in a sentence for your report. *Note the bus connections coming from the MicroBlaze; is it a Von Neumann, "pure Harvard", or "modified Harvard" machine and why?*

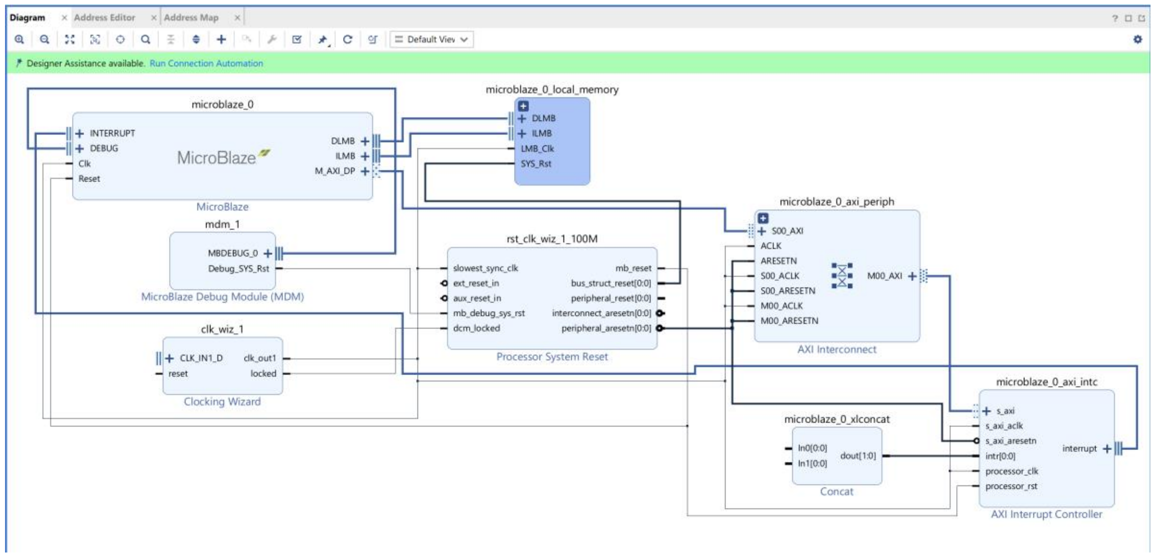
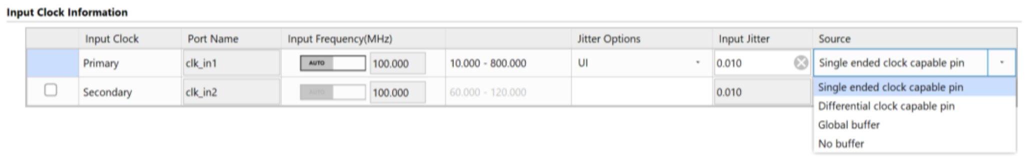


Figure 3 - Default MicroBlaze Microcontroller Preset

Finally, the default clock which the preset expects is a differential clock (we will discuss differential signaling in class, and it is a technology which enables many of our high-speed data links such as USB and HDMI). However, for our purposes, we will use a single-ended clock (the same 100 MHz clock you have been using), so this needs to be changed. Double click on the "Clocking Wizard" block and change "Differential clock capable pin" to "Single ended clock capable pin" as shown in Figure 4:



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	

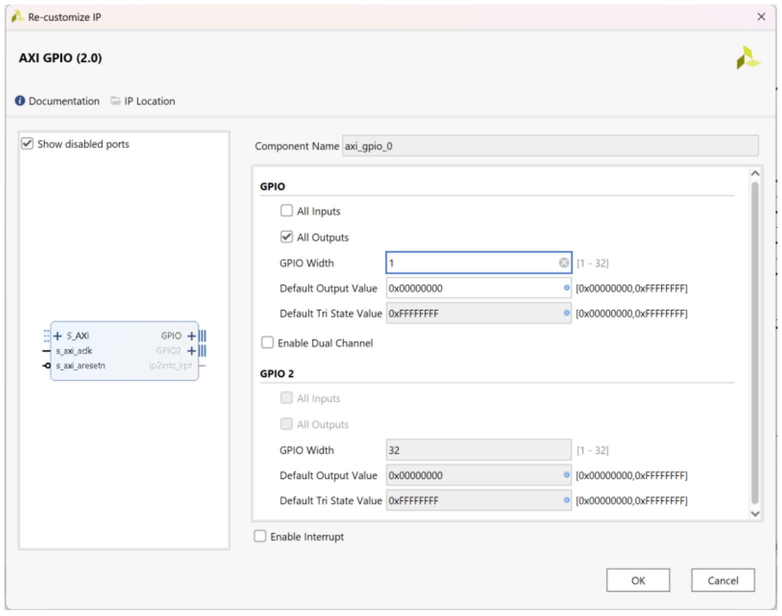
Single ended clock capable pin
Differential clock capable pin
Global buffer
No buffer

Figure 4 - Changing the Clocking Wizard

Adding Additional Peripherals to the Block Design

Although we have created a bare-bones MicroBlaze system, we still have no way of really interacting with the system yet – we need to add some peripherals. We will primarily use two peripherals, a GPIO (General Purpose I/O) to control an LED, as well as a UART (Universal Asynchronous Receiver Transmitter) to give us some basic printf support for debugging.

Go ahead and add one each of the following peripherals to your existing system:
AXI GPIO and AXI Uartlite. Typically speaking, the peripherals will need to be configured to be useful. In the case of the AXI GPIO, you should set up the peripheral (for now) to just have a single output used to blink a single LED. You can access the parameters associated with the peripheral by double clicking on the module in the block design. Reconfigure the GPIO as follows:



Re-customize IP

AXI GPIO (2.0)

Documentation   IP Location

☑ Show disabled ports

Component Name axi_gpio_0

GPIO

☐ All Inputs
☑ All Outputs

GPIO Width	1	[1 - 32]
Default Output Value	0x00000000	[0x00000000,0xFFFFFFFF]
Default Tri State Value	0xFFFFFFFF	[0x00000000,0xFFFFFFFF]

☐ Enable Dual Channel

GPIO 2

☐ All Inputs
☐ All Outputs

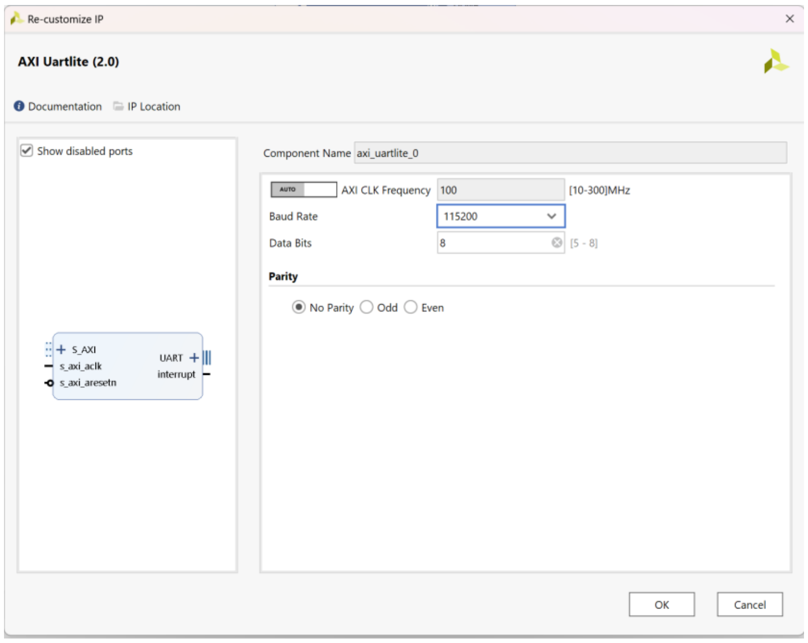
GPIO Width	32	[1 - 32]
Default Output Value	0x00000000	[0x00000000,0xFFFFFFFF]
Default Tri State Value	0xFFFFFFFF	[0x00000000,0xFFFFFFFF]

☐ Enable Interrupt

OK   Cancel

Figure 5 - GPIO Configuration

The UART (Universal Asynchronous Receiver Transmitter) should also be reconfigured. The only thing to change here is the baud rate (to 115200, or 115.2kbaud). This is fast enough to transmit about 10 Kbytes per second, extremely slow by modern standards, but fast enough to transmit console text for the purposes of debugging (e.g., printf or scanf). *What does the "asynchronous" in UART refer to regarding the data transmission method? What are some advantages and disadvantages of an asynchronous protocol vs. a synchronous protocol?*



Show disabled ports: ✓

Component Name: axi_uartlite_0

AXI CLK Frequency: 100 [10-300]MHz

Baud Rate: 115200

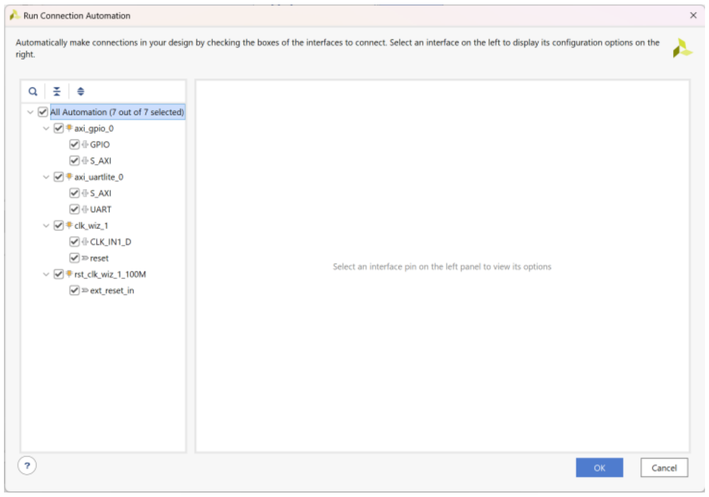
Data Bits: 8 [5 - 8]

Parity

[✓ No Parity] Odd Even

Figure 6 – AXI Uartlite Configuration

At this point, your two new peripherals will be configured, but still unconnected to the rest of the MicroBlaze AXI bus. Typically speaking, Vivado will be able to assist you in making the connections automatically (by using the Run Connection Automation macro). You can do this now by clicking the "Run Connection Automation" macro on the top left of the diagram. You should select all the available options, as shown in Figure 7.



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.

- ✓ All Automation (7 out of 7 selected)
  - ✓ axi_gpio_0
    - ✓ GPIO
    - ✓ S_AXI
  - ✓ axi_uartlite_0
    - ✓ S_AXI
    - ✓ UART
  - ✓ clk_wiz_1
    - ✓ CLK_IN1_D
    - ✓ reset
  - ✓ rst_clk_wiz_1_100M
    - ✓ ext_reset_in

Select an interface pin on the left panel to view its options

Figure 7 - Connection Automation

However, the default MicroBlaze template you are using has a “Concat” e.g., concatenate module on the input to the interrupt controller. This combines the interrupt signal from multiple sources together, while you only have a single peripheral which will generate an interrupt (the UART module). Therefore, you will need to delete the “Concat” module – and manually connect the interrupt output from the UART to the interrupt controller via the `intr[0:0]` port. The final connected MicroBlaze setup appears in Figure 8 below. Note that if you create another device which uses interrupts, you will need to re-create the “Concat” module to properly concatenate the interrupt signals together. *You should have learned about interrupts in ECE 220, and it is obvious why interrupts are useful for inputs. However, even devices which transmit data benefit from interrupts; explain why. Hint: the UART takes a long time (relative to the CPU) to transmit a single byte.*

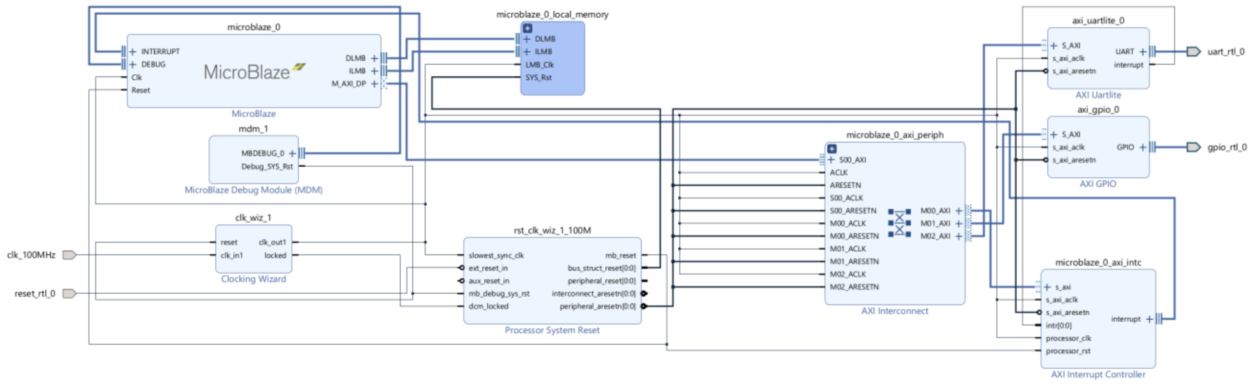


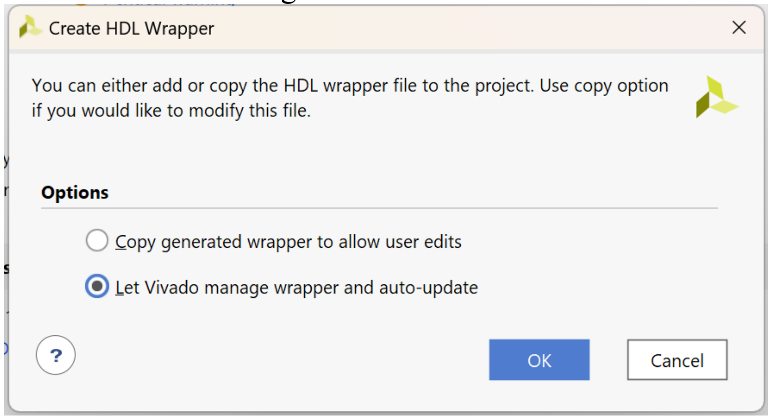
Figure 8 - MicroBlaze Connected

Notice that the Connection Automation created a small number of input ports (`clk_100MHz` and `reset_rtl_0`) and a small number of output/bidirectional ports (`uart_rtl_0` and `gpio_rtl_0`). The inputs are self-explanatory, although be aware that by default the resets within a MicroBlaze system are **active low,** though this may be reconfigured in the Processor System Reset module. The outputs are just the GPIO output which you will use to control the LED, as well as the UART, which consists of two connections – an output called `uart_rtl_0_rxd` (which denotes data entering the UART from the connected computer, and `uart_rtl_0_txd` (which denotes data exiting the UART and being displayed on the connected computer).

Also, switch to the Address Editor tab and make note of the contents. Depending on the order you added the modules, the addresses assigned to each module might be different. Note that by default, the GPIO and UARTlite modules take up 64Kbytes of address space, even though they do not require nearly this many addresses. Since you are not limited by address space in this application, we will use the default settings for now. Also, note the base addresses of the various peripherals, specifically the GPIO peripheral – as you will need it later. You can also see this information presented graphically in the Address Map tab.

Finally, right click somewhere (but not on a module) in your diagram tab and select “Validate Design”. This will ensure that all the connections which were made are valid.

Save your design and close out the block design editor. Your project should now have a Design Source called “mb_block” which shows up in orange, indicating it is a block design. Typically, when you are working with a block diagram, you will want to create the HDL wrapper which will hold your module instantiation for the block. Right click on “mb_block” and select “Create HDL wrapper”. Then select the following:


Create HDL Wrapper

You can either add or copy the HDL wrapper file to the project. Use copy option if you would like to modify this file.

Options

Copy generated wrapper to allow user edits [✓ Let Vivado manage wrapper and auto-update]

Figure 9 - Create HDL Wrapper

The HDL wrapper is a Verilog file which has only a **module instantiation** of our MicroBlaze system. Normally you will want to start with the wrapper and modify it accordingly (e.g., instantiate other logic not part of your block design). However, Vivado does not support creating SystemVerilog HDL wrappers (only Verilog and VHDL). Therefore, you will instead use the provided SystemVerilog top-level file (mb_intro_top.sv), which you will modify as necessary.

Note that although you are provided with a top level for this lab, you should always have Vivado create an HDL wrapper for your block designs, even if you will not use it directly. This is because it provides a good reference for all the input and output names to the various components inside your block design (e.g., the output of your GPIO is called “gpio_rtl_0_tri_o”, which would be challenging to figure out without referencing the HDL wrapper).

You can now return to Vivado and import the provided top level file (mb_intro_top.sv). This is a minimal top-level file which instantiates the block design you just created. Make sure that the file is set as your top-level (e.g., it shows up in **bold**). Import the proper pin assignments, and then generate the bitstream for the design. If there are any issues with the synthesis or implementation, double check that the signal names in the top-level file reflect the proper inputs and outputs from the block design (this is the most likely source of errors).

Once the bitstream is generated, you do not need to program as before. In theory we could, but we will instead program the FPGA through Vitis SDK at the same time as the MicroBlaze code to avoid confusion about whether the FPGA is programmed or not. Instead go to File->Export and Export the hardware:

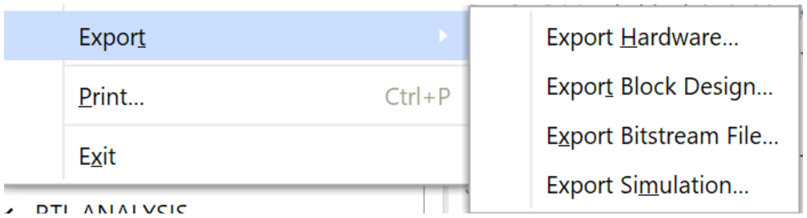
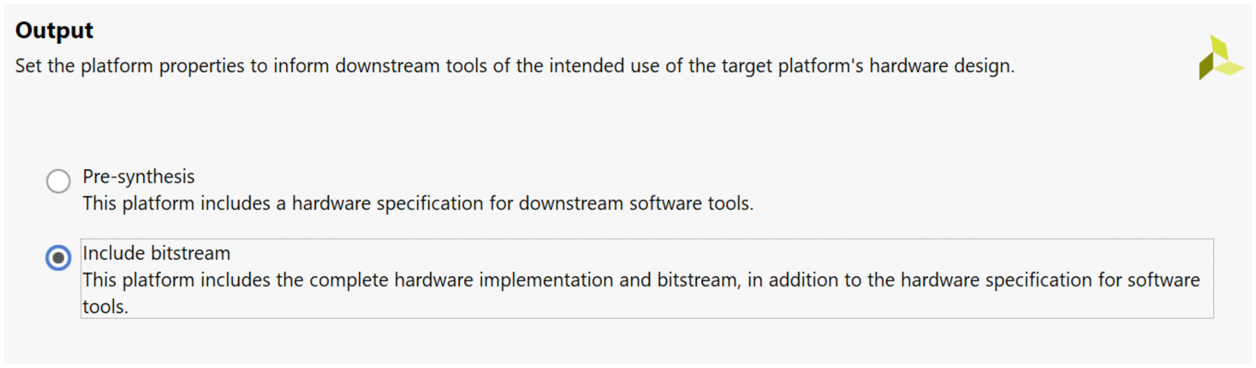


Figure 10 - Export Hardware

Click next and make sure you select "Include Bitstream" in the following screen:



Output

Set the platform properties to inform downstream tools of the intended use of the target platform's hardware design.

○ Pre-synthesis
This platform includes a hardware specification for downstream software tools.

● Include bitstream
This platform includes the complete hardware implementation and bitstream, in addition to the hardware specification for software tools.

Figure 11 - Make sure you "Include Bitstream."

On the next screen, make sure you Export to your project directory. Hopefully your project path is short (see note at the beginning of this document) and click Finish. The dialogue will now close, but now you can Launch Vitis IDE from the Tools menu in Vivado.

Software Setup:

When you first launch Vitis, it will ask for a "Workspace" location. You will want to create a new workspace subdirectory (in Windows or Linux) located in the project directory itself, e.g.: "C:\Users\ljames23\ece385\labx\workspace" if using an EWS machine. **Do not use the default path**, as it will dump all your software projects into the same default directory – which will cause a multitude of path, file association, and organizational issues when you have projects from multiple unrelated labs in the same workspace. You can then go ahead and press "Launch". Vitis IDE is the SDK or Software Development Kit for software associated with Xilinx/AMD FPGA products. In this course, you will only use it for developing software for MicroBlaze, but it may also be used for software development for embedded ARM cores in devices which have such cores.

You will need to make a new project, based on the MicroBlaze system you have already created and exported. To do this, you will go to File->New->Application Project. Click through the splash screen, and you will get to a screen where it asks for a platform. Depending on the exact options you have installed, you may have some pre-built platforms for other FPGA boards. However, we will instead use the platform you previously exported, so switch the top tab to "Create a new platform from hardware (XSA)".

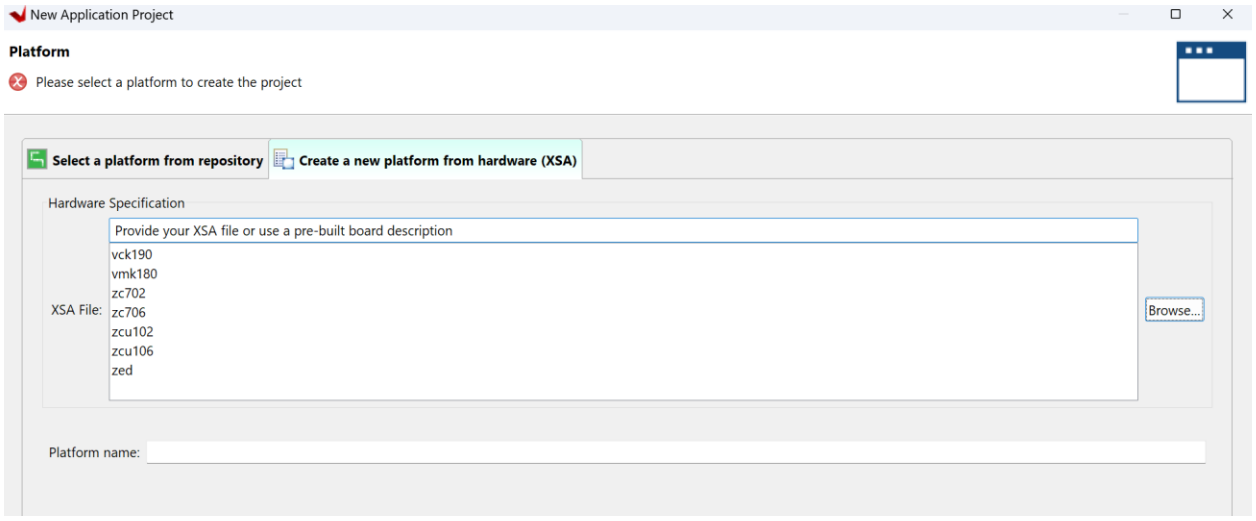

New Application Project
Platform
Please select a platform to create the project
Select a platform from repository
Create a new platform from hardware (XSA)
Hardware Specification
XSA File:
Provide your XSA file or use a pre-built board description
vck190
vmk180
zc702
zc706
zcu102
zcu106
zed
Browse...
Platform name:

Figure 12 - Import XSA

As expected, here you should click Browse and point Vitis to the previously exported XSA. You can leave it as the default platform name, which will just be the same name as the provided top-level module and click next.

Afterwards, you will give your project a name. You can give it any name, but the name should be short and not include any unusual characters (e.g., spaces or Unicode). In my case, I simply called it "mb_blink", as that's all our provided code will do (blink an LED). It will also give you the option to associate your project with a "system project" – this is what Vitis calls the BSP (Board Support Package). We will discuss what the significance of this project is in lecture so for now, just click Next. It will then ask you for a domain, which is a way of organizing larger projects together (for example, if you wanted to use Linux, you will need to create a domain which includes the customized Linux kernel). In our case, the defaults are fine. Note that our project is a "standalone" project, which implies that we have no operating system, but rather are executing the code directly on MicroBlaze – so you can click Next here as well.

Finally, you will be given the choice of several project examples. Since you're already provided some code (which you will need to study to understand and extend for this lab), choose "Hello World". Once this is completed, you will be returned to Vitis, but with 2 new projects on the left "Explorer" panel.

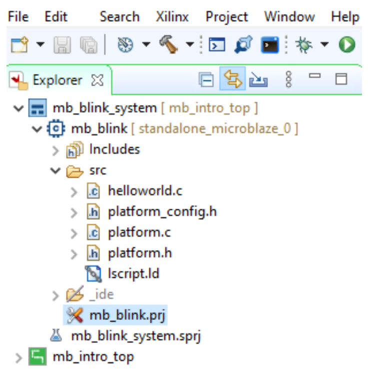


Figure 13 - Explorer Panel

Notice that there are 2 projects with different icons. You should have your software project, with the blue icon and your system itself, with the green icon. Furthermore, within the software project, you have the application itself (called mb_blink), plus some provided sources. Here you can populate your sources, just like a typical IDE (Visual Studio, Android Studio, Eclipse, etc..)

Import the provided C file (mb_blink.c) into the project (e.g., right click on the project, and click Import Sources). In addition, delete from the project the pre-provided hello_world.c file (to prevent conflicting with the main() function provided in mb_blink.c). Note that the project will still have syntax errors as you still need to fill in the memory address of the LED GPIO peripheral. This is conceptually identical to the MEM2IO address (0xFFFF) from the SLC-3 lab, but the actual address will be different (depending on the order you added the peripherals, etc.). One way to find this address is to go back to Vivado, open the block design, and go to “Address Map”.

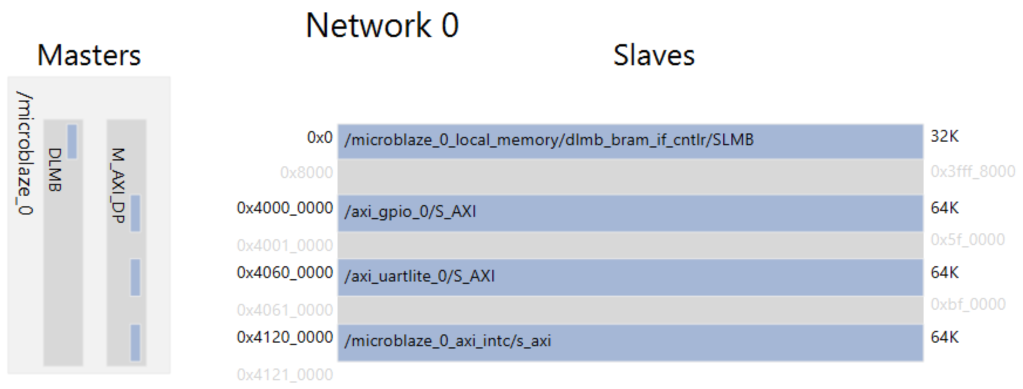
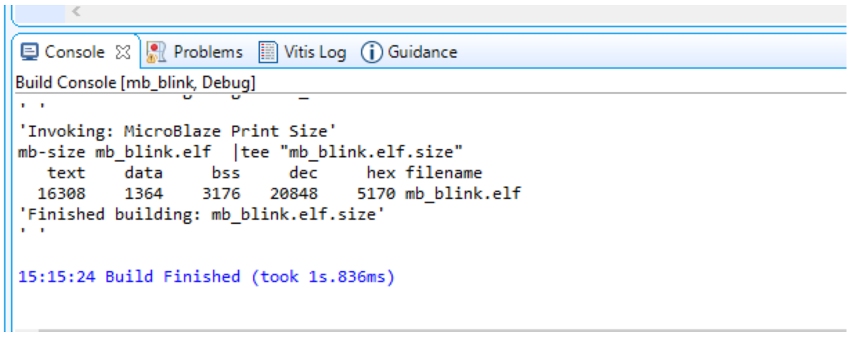


Figure 14 - Example Memory Map

Here, you will see the memory map for both the instruction and data buses. *Why are the UART and LED peripherals only connected to the data bus?* In this case, the LED GPIO is mapped to address 0x40000000. Modify the provided code accordingly. Alternatively, you can use a define directly from a file called “xparameters.h” (provided by the system project) to automatically populate this address. If you like, examine the contents of the file, and use the automatically populated define instead, which avoids having “magic numbers” in your code. Note also that the provided code calls “init_platform()” before going into the main loop. This is used to do things like initializing the MicroBlaze caches and the serial port configuration.

You must be able to explain what each line of this (very short) program does to your TA. Specifically, you must be able to explain what the volatile keyword does (line 18), and how the set and clear functions work by working out an example on paper (lines 30 and 33).

Your code should now pass syntax checks. Build the project (under the Project menu) and fix any syntax errors which might arise. Examine the build log in the Console tab (if you do not see the proper text in the console, click on the mb_blink project under Explorer).



```
'Invoking: MicroBlaze Print Size'
mb-size mb_blink.elf  |tee "mb_blink.elf.size"
   text    data     bss     dec     hex filename
  16308    1364    3176   20848    5170 mb_blink.elf
'Finished building: mb_blink.elf.size'
' '

15:15:24 Build Finished (took 1s.836ms)
```

Figure 15 - Executable size

Look at the various segments (text, data, bss), what does each segment mean? What kind of code elements are stored in each segment? Also note the size of the executable in bytes. Remember that we configured 32Kbytes of on-chip memory to use as our program memory. *Why does the provided code, which does very little, take up so much program memory? Hint: try commenting out some lines of code and see how the size changes.*

You are now ready to test and debug your software program. Switch Vitis to the debug view by clicking on the top right. One unusual thing about Vitis is that many tasks (e.g., text editing) may be done in either design or debug view, but each view has a different default layout to optimize for their respective tasks.

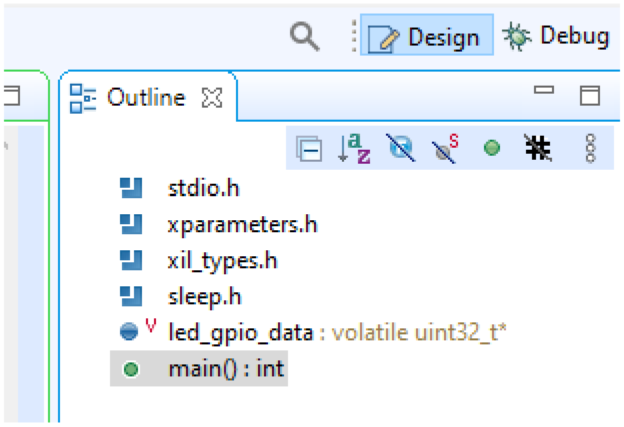
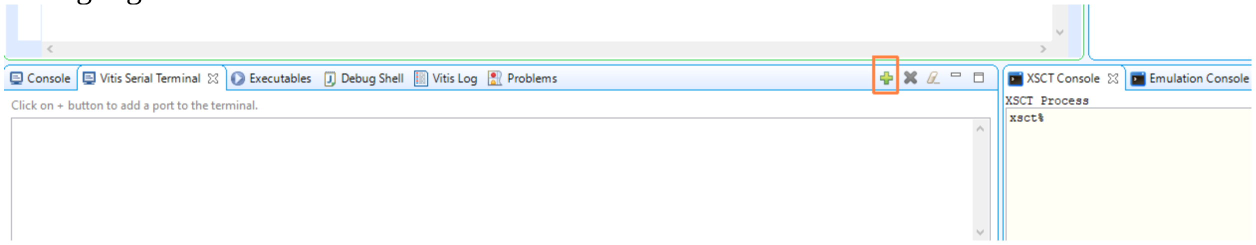


Figure 16 - Vitis Debug View

Notice that Vitis should be quite familiar to students who have encountered the Eclipse IDE before – in fact, it is a port of Eclipse customized for Xilinx FPGA products. In fact, you can use all the tools you are already familiar with for C/C++ development here, including access to GDB to set breakpoints, examine memory, etc.

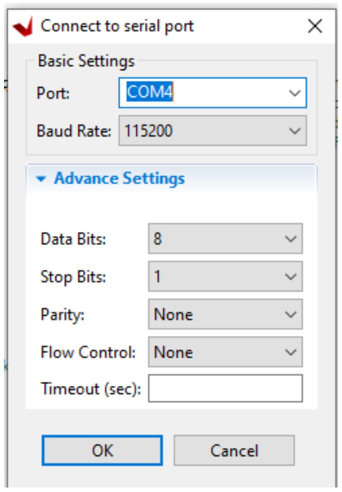
One thing to note is that the provided code has some built-in printf statements. Normally when you are developing a console program, you would expect this to print to a text console, (e.g. Windows cmd prompt, or Linux shell). Since we are executing this code on MicroBlaze, the printf statements will be redirected to a serial terminal. Connect to the serial terminal by pressing the highlighted connect button.



Click on + button to add a port to the terminal.

```
XSCT Process
xsct%
```

Figure 17 – Vitis Debug Screen



Connect to serial port

Basic Settings

Port: COM4

Baud Rate: 115200

Advance Settings

Data Bits: 8

Stop Bits: 1

Parity: None

Flow Control: None

Timeout (sec):

Figure 18 - Serial port connection

The exact port may vary from system to system (and it may be called `ttyUSBn` on Linux instead of `COMn` on Windows). Typically, this is the highest port number (assuming the FPGA board is powered up that it is the most recent connected device). The other parameters correspond to what you chose during the UARTlite configuration. After you have connected to the serial port console, printfs from the MicroBlaze code will show up under the “Vitis Serial Terminal” tab.

If you are on Linux and the port menu is blank, you should open a terminal and type in the following commands: `sudo chmod 666 /dev/ttyUSB0` followed by `sudo chmod 666 /dev/ttyUSB1`. This will ensure that Vitis has permission to access the serial terminal. On the Urbana board, the debug serial port comes up as `ttyUSB1`.

We need to set up a “run configuration” to actually test the code on the hardware as this involves uploading the “.elf” file to the hardware and starting the execution of the MicroBlaze. Click the down arrow to the right of the green (play) button (run button) to expand the menu and click Run Configurations.

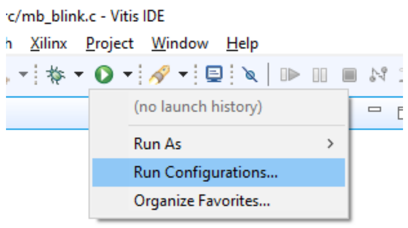
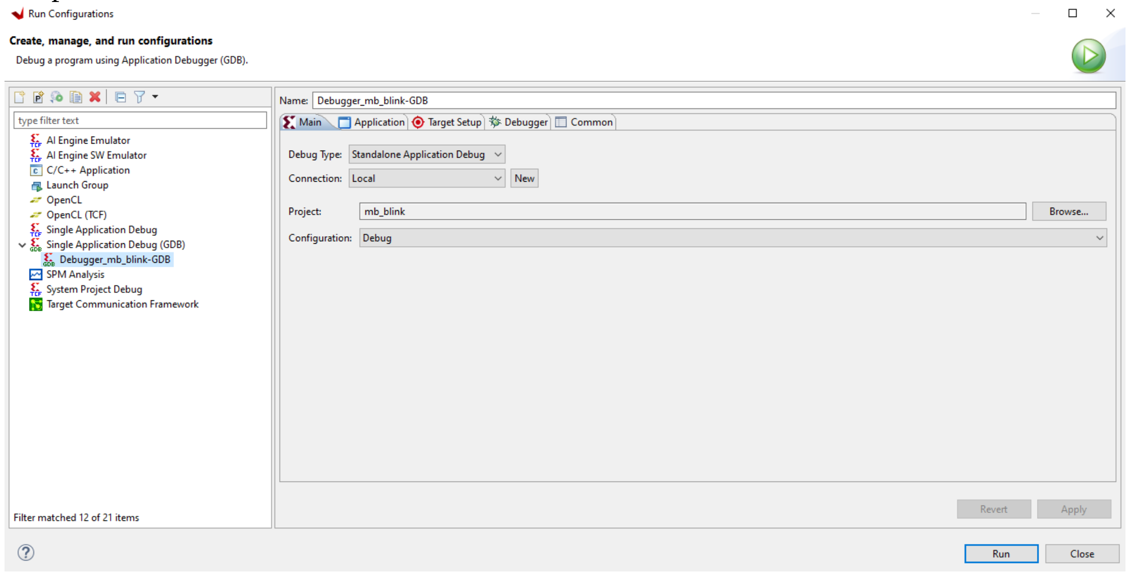


Figure 19 - Run Configuration

Select "Single Application Debug (GDB)" which will use GDB as our debugger and double click on this option:



Name: Debugger_mb_blink-GDB

Debug Type: Standalone Application Debug

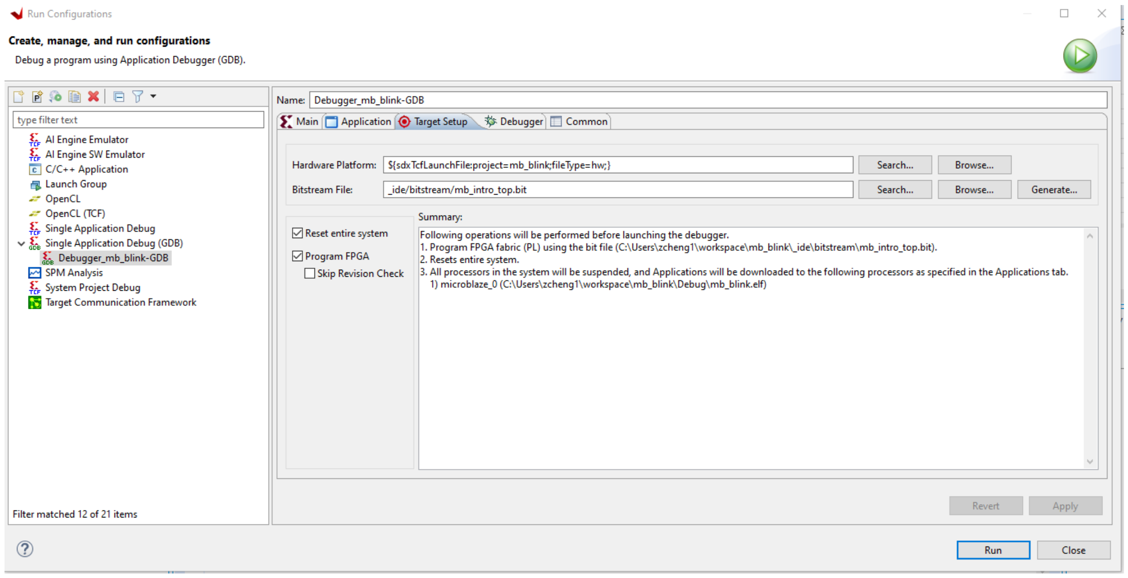
Connection: Local

Project: mb_blink

Configuration: Debug

Figure 20 - Run Configuration Panel

For the most part, the defaults here are fine, but if you have multiple projects (which you may for the second part of the lab), you may select which project to run here. Also verify that "Reset entire system" and "Program FPGA" are checked in the "Target Setup" tab.



Run Configurations

Create, manage, and run configurations

Debug a program using Application Debugger (GDB).

- AI Engine Emulator
- AI Engine SW Emulator
- C/C++ Application
- Launch Group
- OpenCL
- OpenCL (TCF)
- Single Application Debug
- Single Application Debug (GDB)
  - Debugger_mb_blink-GDB
- SPM Analysis
- System Project Debug
- Target Communication Framework

Filter matched 12 of 21 items

Name: Debugger_mb_blink-GDB

Hardware Platform: ${sdxTcfLaunchFile:project=mb_blink;fileType=hw;}

Bitstream File: _ide/bitstream/mb_intro_top.bit

[✓] Reset entire system

[✓] Program FPGA

[ ] Skip Revision Check

Summary:

```
Following operations will be performed before launching the debugger.
1. Program FPGA fabric (PL) using the bit file (C:\Users\zcheng1\workspace\mb_blink\_ide\bitstream\mb_intro_top.bit).
2. Resets entire system.
3. All processors in the system will be suspended, and Applications will be downloaded to the following processors as specified in the Applications tab.
   1) microblaze_0 (C:\Users\zcheng1\workspace\mb_blink\Debug\mb_blink.elf)
```

Figure 21 - Target Setup

This will automatically program the FPGA bitstream and then transmit the ".elf" to the on-chip memory and execute it. Note that in the future, you do not need to create a new run configuration, and you can just "Run As" with the configuration you just created or press the green play button.

<u>IMPORTANT</u>:
Whenever the hardware part is changed, it needs to be compiled in Vivado and **re-exported**. Refer to Figure 10 and go through the process of exporting the hardware with bitstream again (make sure the runs in Vivado have all completed and the bitstream has been generated successfully first). Make sure to override the existing file.

Afterwards it is essential to update the system project in Vitis. If you had closed out of Vitis prior to your Vivado build, sometimes Vitis will automatically detect that the hardware has changed and prompt you to update when you open it back up. However, if Vitis was running in the background it will typically not detect a change automatically. In Vitis, you can force the update by simply right clicking on the system project and clicking Update Hardware Specification. Then click OK on the next screen. You then need to go to Project-> Clean (select everything), and Project->Build All.

If you neglect this step, you will have software which may be compiled expecting different hardware (e.g., different memory maps or previous versions of your custom IPs). This will cause bugs which are very difficult to catch.

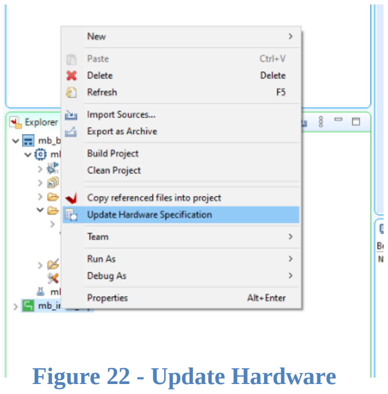


Figure 22 - Update Hardware Specification

Click ***Run*** to run the program. Your LD0 should start blinking on your board as soon as the binary has been transmitted. Addition, text corresponding to the status of the LED should show up in the serial console. Note that to do the actual lab assignment, you must add another GPIO block as input corresponding to the switches. Do not forget you must connect the new GPIO to your MicroBlaze CPU, modify the top-level SystemVerilog, assign the correct pins and update the hardware platform. The documentation of the “Xilinx AXI GPIO” will be useful for this portion of the lab: Xilinx AXI GPIO. *Make sure you understand the register map on page 10. If the base address is 0x40000000, how would you access GPIO2_DATA (for example?).*