­A screenshot of a cell phone

Description automatically generated

Vivado SDK Basic IP Integrator

Benjamin A. Blouin

*Benjamin.blouin@temple.edu*

**Summary**

Embedded development on the Digilent Zybo board is achieved through processing in the Vivado Xilinx Zynq Processor System (PS) and Programmable Logic (PL), using predefined intellectual property (IP). After the PL is designed to the design characteristics, the Vivado SDK allows the PS and PL to communicate over the S-AXI bus architecture. Using on-board 4bit LEDs and push-buttons, each push button press-depress is used to trigger a unique behavior. On pressing the button, the current LED output is held until the button is depressed. The button choice has one or four behaviors: counting-up in binary from zero to 15; counting down in binary from 15 to zero; displaying from zero to , then displaying the truncated finite arithmetic sequence until 0; displaying the following repeating decimal sequence in binary: 9, 6,10, 5, 12,…, with all cases performing in perpetuum. The Verilog and the C language is used to implement the design behavior, whilst using Zynq Book Tutorials and given source files as a template. This lab successfully introduces the development board, environments, and was successfully completed satisfying all design requirements.

**Introduction**

This lab is designed to introduce us to the Zybo development board, and the accompanying Xilinx software tools. Vivado is used to design the PL, and the Xilinx SDK is used to program the PS. The PS is used, in this lab, controls the behavior of the output LEDs based on the push-button pressed. The project assumes knowledge of general FPGA design and C coding from pre-requisite classes. It also assumes you have already installed the required software on one’s personal laptop. The lab requires four unique behavior cases, with a fifth minor case. The report provided describes: the creation of a Vivado project, implementation of Xilinx IP into the PL, the creation of an SDK project, C programming coding for specific behavior analysis, programming the Zybo board, real-world verification, and any troubleshooting involved.

**Discussion**

**Vivado Project Design**

Vivado project design is straightforward, as we did not have any external constraints, and because the Verilog coding was performed automatically by Vivado. Creation of the block design was through the user interface, as was implementing IP. My initial design had to be revised because I used one GPIO block, but the lab required two separate GPIO blocks. After verifying the block design with the built-in tools, an HDL wrapper for the design was created with Vivado managing any updates to the design. Each step in the Vivado IDE design process after block design can be automated by asking Vivado to create a bitstream. A popup dialogue asks if doing the intermediate steps automatically is wanted, and after accepting the process takes a fair amount of time to synthesize and implement the design. This is an immensely powerful tool, as all we have done is create a block design, but the software does all the circuit routing, tests timing, and a myriad of other hardware-based steps that are critical to a functioning PL. At this point we have designed the PL hardware which will be exported, including the bitstream, to the SDK. The Vivado IDE can open a workspace that is in the same project folders as the original Vivado project.

**Xilinx SDK Design**

The Xilinx SDK is used to program the PS to interface with any PL designs created previously. The hardware design is included by Vivado hardware exportation. The SDK creates a project skeleton, which has yet to have a program or operating system bootstrapped to the PS. Creating a new, blank standalone project creates the minimum necessary files to program the PS; this program would be blank, but technically runnable.

The lab suggests importing the example source file corresponding to a previous exercise to use as a template. The lab also gives suggested code that includes some preprocessor definitions, GPIO initializations, and GPIO data direction settings. They are by no means the necessary code to complete the project, and by themselves cannot even be compiled. The lab requires implementation of the code given, example code from classroom tutorials, and most importantly my coding abilities.

**Coding Descriptions**

The C language conventions are used, with very important addition of some key header files that are part of the Board Specific Package (BSP). The included header files include:

1. xxparameters.h, which is the BSP starting point. It represents an abstraction layer, for instance defining the CPU\_ID as 0U. This designation would be pedantic to implement from scratch. It defines platform specific definitions, sleep timer configuration, RAM low/high addressing, DEVCFG drivers, and a host of other BSP specific references.
2. Xgpio.h, which is described in the file comment introduction, “This file contains the software API definition of the Xilinx General Purpose I/O (XGpio) device driver.
3. Xstatus.h, which is described in the file comment introduction, “The xstatus.h file contains the Xilinx software status codes. These codes are used throughout the Xilinx device drivers.” (Comment cleaned up for clarity)

Preprocessor defines are used to simplify GPIO device identification, as well as defining the software delay length. To GPIO device driver instances are created through the included XGpio method. From the xgpio header file, “The user is required to allocate a variable of this type for every GPIO device in the system. A pointer to a variable of this type is then passed to the driver API functions.” The struct includes: device base address, device ready flag-type variable, whether interrupts are present in hardware, and where there are one or two channels.

A function prototype is created, board\_variable\_init(); this function was created to separate the GPIO initialization and data direction register settings, to be described later.

Three global variables are created which represent the pressed button identification, the status of the button (pressed/depressed), and a delay variable. All these variables are static, which keeps the variables value between invocations, or to restate, it preserves its value even after they are out of scope, as well as limiting the scope to the file for which it was declared. This is useful for when we want a variable to be immutable in memory except when it is specifically asked to be changed. They are allocated a memory in a date segment, and not the stack. They are pre-initialized to 0 if not manually initialized. All these qualities are warranted, as we would not want a variable representing the button’s value to be lost due to an issue with the stack, or when the variable is out of scope.

Non-static variables created include the led counting variable, the fourth cased starting variable, and a third case sentinel used to determine LED travel direction.

Then the board initialization method is called. Two methods are used, XGpio\_Initialize and XGpio\_SetDataDirection. The initialization method takes as arguments the previously created XGpio instance pointer, passed by reference, as well as a device ID. The configuration creates a struct the include: the Unique ID of device, device base address, where interrupts are supported in hardware, and whether one or two channels are enabled. It looks in the device configuration if the user is trying to implement an actual piece of hardware.

The data direction set method expects the XGpio instance pointer, a channel integer, and a direction mask. The direction mask specifies discrete bits zero as output and one as input. In the cases for both our GPIO devices, they are both one channel, with the LEDs as outputs (0x00) and the buttons as inputs (0xFF). This completes board initialization.

**Behavior Logic Implementation**

There is an important behavior to control for that must work before any other logic can be successful. The board debounces button presses, but we must determine whether a button is pressed for logical reasons. If a button is pressed, and is not released, the current value output to the LEDs is held. If the button is released, the buttons value represents a behavior selection. It is therefore, necessary to determine if a button has not been pushed, which is represented as a button identification as zero. The Xgpio\_DiscreteRead method takes in the XGpio instance pointer, and the channel, and returns an unsigned 32bit integer. In the case of the buttons only the first four bits are needed. Each button represents a single bit flip, in decimal represented by 1,2,4, and 8. Any other case doesn’t represent a behavior requirement and is caught by the switch statements default case. Also, the zero value represents a button not being pressed, which has a behavior found in each actionable case.

When a button is pressed, the value is saved in a static variable. This variable is then tested for being non-zero. A non-zero value is saved in a switch-statement argument variable; it represents when the user wants something to change. Change can also include restarting the current behavior. In this controlling if-statement, the behavior case variable for counting and patterns are reset, as required by the lab instructions.

There are four behavior cases, which are implemented with a switch statement. This was used because there are a finite number of cases, which lends itself to being easily debuggable; if something is wrong with a case, I only need look in the case statement block to determine problems.

Something to be explained before the case explanations is that another check is needed to make sure the button has been depressed. The same type if-statement is used before; if the button is depressed the coded behavior is implemented, but if the button is still pressed the switch statement is exited. This means that the LED output is not updated, which is a lab behavior requirement.

This is also an appropriate time to describe the XGpio\_Discrete Write method, which expects the arguments for the XGpio pointer, the channel, and an unsigned 32bit integer representing data. The write method doesn’t return any value.

The first case is a simple binary counting ascension starting from zero, updating the LED value, then updating the data variable.

The second case mirrors the first, with a descending count. This count also starts from zero, meaning the same starting value as the first case can be used. This means the same variable can be used.

The third case is describe in the lab as a bar graph. Starting from zero a bit is shifted into the LSB. On the next update, the original bit is again shifted left, with another bit shifted into the LSB. Once all four bits (LEDs) are lit, the process is reversed, with a zero shifting into the MSB until zero is reached again. In decimal this represents the sequence: 0,1,3,7,15,7,3,1,0. The ascending case can be represented by the summation . The descension can make use of the way the C-language does integer division.

Any remainder is truncated, so 15 divided by two in floating point math would result in 7.5, but in integer division the result is seven. This continues for all the values: So on ascension is a summation, and on descension is a simple division. A sentinel variable is used to determine number output direction, with zero and 15 being the testable values.

Case four represents a repeating pattern, represented in decimal as: 9,6,10,5,12,3. A set of if statements completes the pattern. Other tactics were tried, for instance, the difference between each two-number set is the partial Fibonacci of 3,5,8, with the difference between pairs being 1,1,2. These look suspiciously like Fibonacci. Also, in binary, the pairs first value can be the complement of XOR of the second value (1001,0110), and vice versa. Since neither pattern implementation seemed easier, a deliberate if-else flow sequence was used.

The final line of code is a for loop. The loop is used as a software delay between output value updates. This allows for human-readable output, with the final count value being the previously defined delay value. The final chosen value of 20000000 balanced readability with throughput.

All the code is encased in an infinite while loop, as no program exit action was defined. This concludes a description of the code, and the program behavior in general.

**Conclusions**

This lab required a user to implement an embedded hardware design with specific output behavior based on user input. The lab required the installation and use of the Xilinx software package, include Vivado and the SDK. It required C-language coding conventions, and board specific method usage. It also required troubleshooting through hardware behavior based on user input and LED output. All objectives were met, and no changes need to be made.

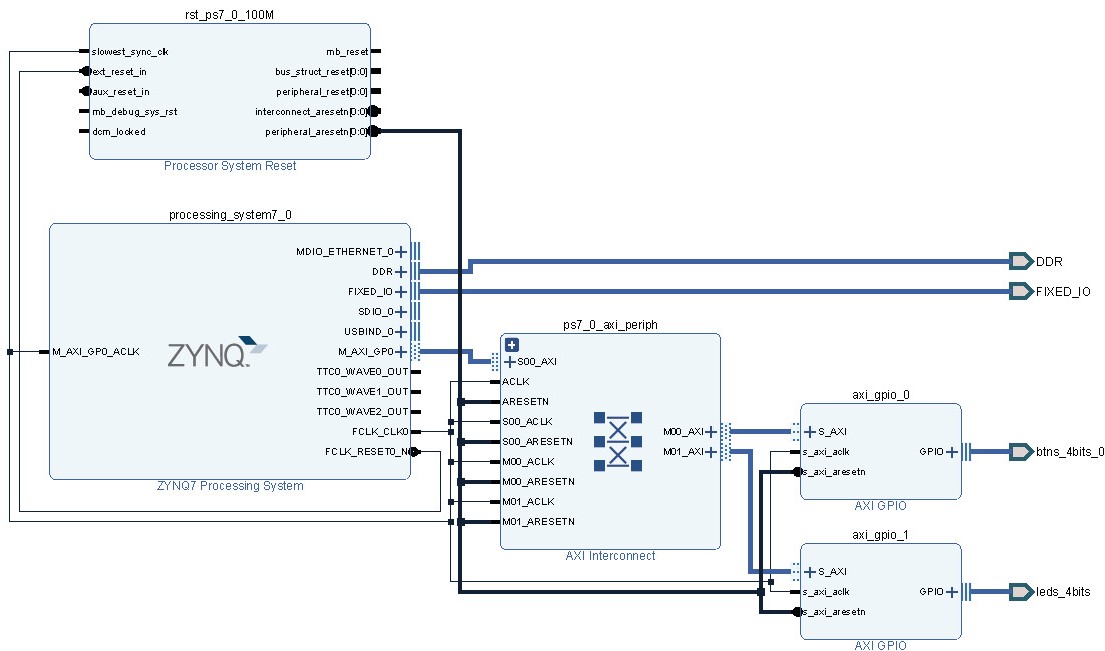
**Appendices**

Figure 1 Block Design

**Code**

A link was used to keep the aesthetic qualities coherent.

[https://github.com/3keepmovingforward3/Embedded-System-Design-Sp19/blob/master/bb\_LED\_BUTTON...](https://github.com/3keepmovingforward3/Embedded-System-Design-Sp19/blob/master/bb_LED_BUTTON/bb_LED_BUTTON.sdk/bb_LED_TEST/src/LED_BUTTON_bb.c)

**Project**

Link to repository of entire project for author attribution integrity.

[Embedded-System-Design-Sp19/tree/master/bb\_LED\_BUTTON](https://github.com/3keepmovingforward3/Embedded-System-Design-Sp19/tree/master/bb_LED_BUTTON)