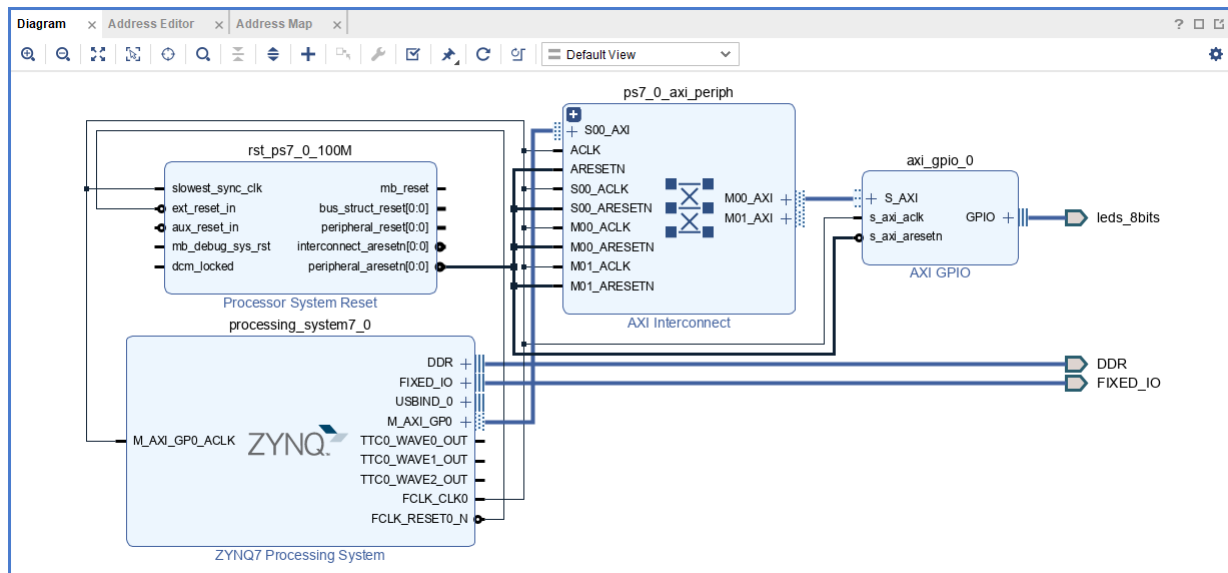


## Introduction

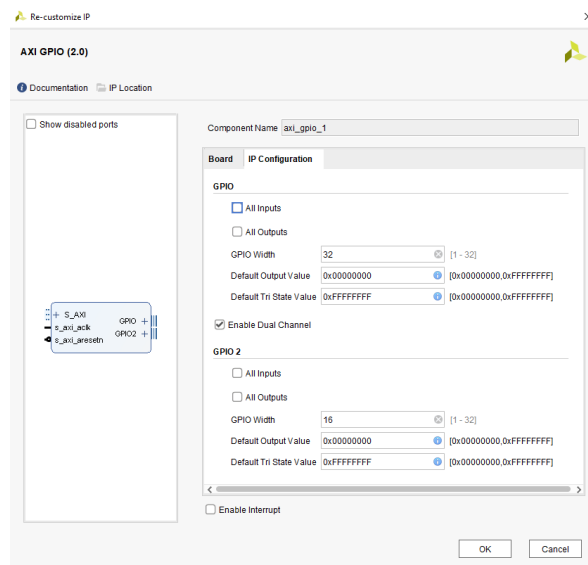
The ZedBoard and Zync processor can be set up to manage many I/O lines. In the previous examples, only a small number of I/O lines were used and their direction (input vs. output) was fixed at design time. In this module we will explore ways to work with I/O while keeping the flexibility of selecting input vs. output from the PS software instead of embedding it in the PL design.

## Design

Create a basic Zync processor block design in the usual fashion. It should have one GPIO IP block dedicated to LED control.



We will add a second GPIO I/O IP block and configure it to manage two banks of lines (32 in one, 16 the other) each for a total of 48 available I/O signals. But before allowing the Connection Automation to run,



double click on the newly added GPIO IP block (or right-click and select “Customize Block”). From the IP Configuration tab, check the box to Enable Dual Channel. Leave all other boxes unchecked and then change the new GPIO Width to 16.

After clicking OK, begin the Connection Automation. After checking the IP to be configured, pick each of GPIO and GPIO2 and in the option box select “Custom” for the part interface. The default is to try and connect to LEDs or buttons on the ZedBoard. We want to define what connector and pins are used. If for some reason the wizard does connect the new GPIO to LEDs or buttons, just select the connection on the diagram and delete. Then re-run the wizard, confirming that Custom was selected.

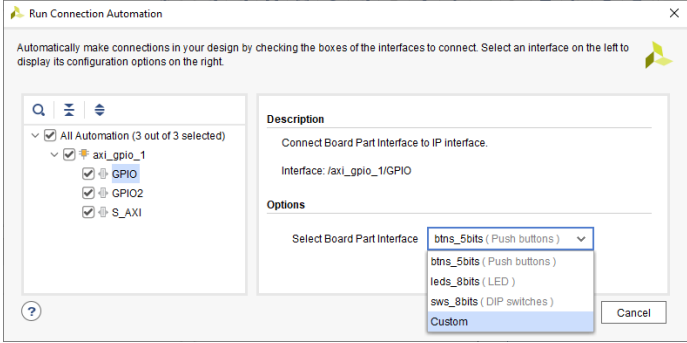
The new connections will have names such as “gpio\_rtl” and “gpio\_rtl\_0”. We will change these to be “GPIO\_A” and “GPIO\_B” using the External Interface Properties dialog usually found just to the left of the diagram.

## Mapping I/O

The newly created 48 bits of I/O need to be mapped to connections on the ZedBoard and have their interface signal standard defined. This is managed inside a constraints XDC file. While these files can be entered manually, it is usually easier to let the Vivado I/O wizard create the basic file and then we make the necessary edits.

From the Flow Manager, select “Run Synthesis”. If there are no significant errors in the design, eventually a dialog box will appear asking for the next step. Open the synthesized design. If the dialog box does not appear after the synthesis is complete (because the “do not show” option was ticked in the past), use the Flow Manager to open the synthesized design.

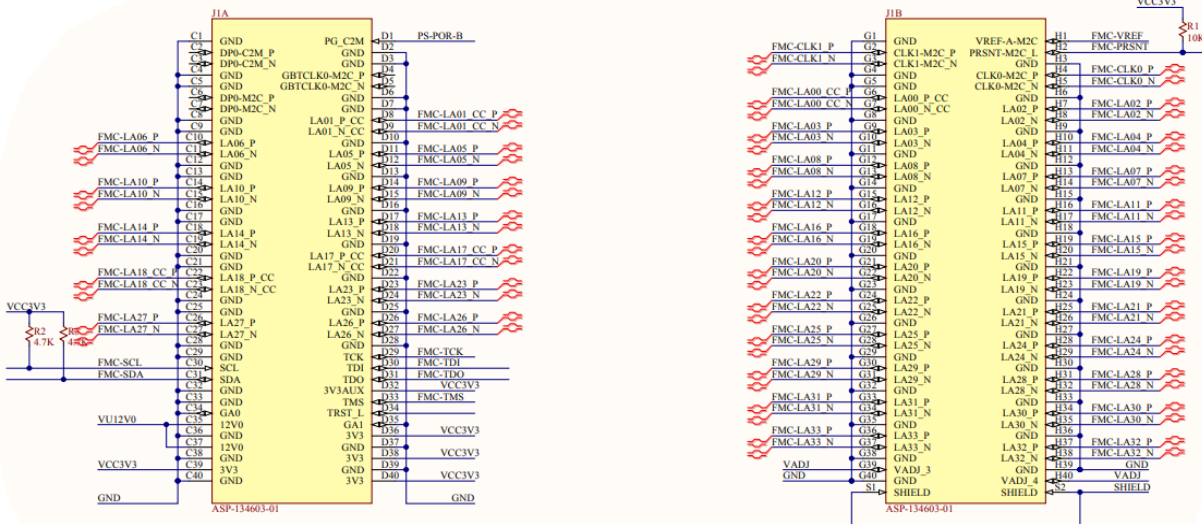
Of interest is the I/O Ports tab. Pins of the FPGA that are not already assigned will generally have the I/O Std column highlighted in red.



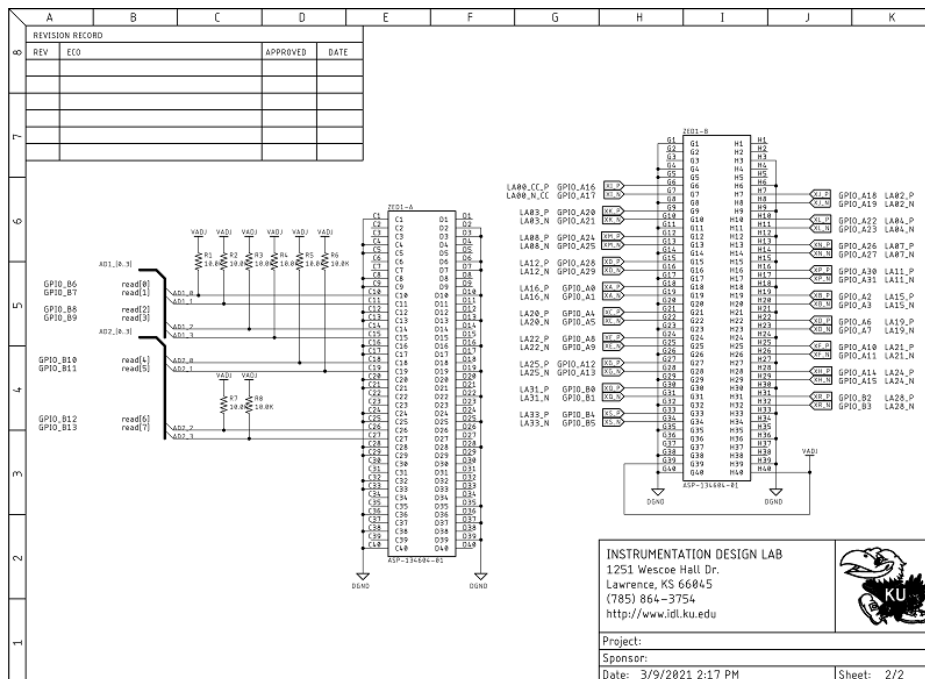
Name	Direction	Board Part Pin	Board Part Interface	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	Stew Type	Pull Type
> DDR_54576 (71)	INOUT					✓	502	(Multiple)*	1.500	(Multiple)	(Multiple)	(Multiple)	NONE
> FIXED_IO_54576 (59)	INOUT					✓	(Multiple)	(Multiple)*	(Multiple)	(Multiple)	(Multiple)	(Multiple)	NONE
> GPIO2_10965 (32)	INOUT					☐		default (LVCMOS18)	1.800	12	12	NONE	NONE
> GPIO_10965 (32)	INOUT					☐		default (LVCMOS18)	1.800	12	12	NONE	NONE
> leds_8bits_54576 (8)	OUT					✓	33	LVCMOS33*	3.300	12	12	NONE	NONE
Scalar ports (0)													

The Fixed column will also show an open tic-box instead of a check-mark for unassigned pins.

In this example, the new 48 I/O lines are to be mapped to I/O's in the FMC (FPGA Mezzanine Card) connector. From the ZedBoard schematic we can find the signal names assigned to the connector.



For example, we might want to use the pin pair LA16\_P (G18) and LA16\_N (G19) as our first two signals on GPIO\_A. They could be used as a differential pair as indicated by the naming convention but in this case, we are going to use them as single-ended I/O. These pin numbers (G18 and G19) correspond to the same pin number in the schematic of a card that will plug into the FMC connector. In the case of the



daughter card  
schematic, G18  
corresponds to a signal  
XA\_P and G19 to XA\_N.  
We want to map XA\_P  
to GPIO\_A[0] and XA\_N  
to GPIO\_A[1].

The next step is to locate the FPGA pin and its corresponding bank assigned to LA16\_P and LA16\_N. The signal names to search for are FMC-LA16\_P and FMC-LA16\_N.

VADJ

<b>BANK 34</b>	
IO_0	H15 XADC-GIO0
IO_25	K15 XADC-GIO1
IO_L1N_T0	K15 XADC-GIO2
IO_L1P_T0	L15 XADC-GIO3
IO_L2N_T0	L17 FMC-LA15_N
IO_L2P_T0	L16 FMC-LA15_P
IO_L3N_T0_DQS	L16 OTG-VBUSOC
IO_L3P_T0_DQS_PUDC_B	K16 PUDC_B
IO_L4N_T0	M17 FMC-LA13_N
IO_L4P_T0	M17 FMC-LA13_P
IO_L5N_T0	N18 FMC-LA11_N
IO_L5P_T0	N17 FMC-LA11_P
IO_L6N_T0_VREF	M16 FMC-VREF
IO_L6P_T0	M15 SW7
IO_L7N_T1	K18 FMC-LA05_N
IO_L7P_T1	L18 FMC-LA05_P
IO_L8N_T1	K22 FMC-LA08_N
IO_L8P_T1	K21 FMC-LA08_P
IO_L9N_T1_DQS	K21 FMC-LA16_N
IO_L9P_T1_DQS	K20 FMC-LA16_P
IO_L10N_T1	K22 FMC-LA06_N
IO_L10P_T1	K21 FMC-LA06_P
IO_L11N_T1_SRCC	K20 FMC-LA14_N
IO_L11P_T1_SRCC	K19 FMC-LA14_P
IO_L12N_T1_MRCC	L19 FMC-CLK0_N
IO_L12P_T1_MRCC	L18 FMC-CLK0_P
IO_L13N_T2_MRCC	M20 FMC-LA00_CC_N
IO_L13P_T2_MRCC	M19 FMC-LA00_CC_P
IO_L14N_T2_SRCC	K20 FMC-LA01_CC_N
IO_L14P_T2_SRCC	N19 FMC-LA01_CC_P
IO_L15N_T2_DQS	M22 FMC-LA04_N
IO_L15P_T2_DQS	M21 FMC-LA04_P
IO_L16N_T2	K22 FMC-LA03_N
IO_L16P_T2	K22 FMC-LA03_P
IO_L17N_T2	K21 FMC-LA09_N
IO_L17P_T2	K20 FMC-LA09_P
IO_L18N_T2	K21 FMC-LA12_N
IO_L18P_T2	K20 FMC-LA12_P
IO_L19N_T3_VREF	M15 FMC-VREF
IO_L19P_T3	N15 BTNL
IO_L20N_T3	M18 FMC-LA02_N
IO_L20P_T3	M17 FMC-LA02_P
IO_L21N_T3_DQS	L17 FMC-LA07_N
IO_L21P_T3_DQS	L16 FMC-LA07_P
IO_L22N_T3	L19 FMC-LA10_N
IO_L22P_T3	K19 FMC-LA10_P
IO_L23N_T3	L18 BTNU
IO_L23P_T3	K18 BTNR
IO_L24N_T3	K16 BTND
IO_L24P_T3	P16 BTNC

IC16D XC7Z020CLG484

Locating the names FMC-LA16\_P and \_N we see they are attached to FPGA pins J20 (FMC-LA16\_P) and K21 (FMC-LA16\_N). So returning to the I/O Ports tab and expanding to locate GPIO\_A\_tri\_io[0] its package pin can be set to J20 and GPIO\_A\_tri\_io[1] to K21. This can be repeated for all the signals on the daughter card as indicated in its schematic. Either by using the graphical entry method or by saving the constraints using the File menu and then editing the file directly via cut-and-paste methods.

Looking at the format we see :

```
set_property PACKAGE_PIN J20 [get_ports {GPIO_A_tri_io[0]}};
set_property PACKAGE_PIN K21 [get_ports {GPIO_A_tri_io[1]}};
```

There is a master XDC file for the ZedBoard available from [ZedBoard.org](http://ZedBoard.org). The appropriate lines could be pasted into our constraints file and edited, changing names like FMC-LA16\_P to GPIO\_A\_tri\_io[0] as needed.

As to the I/O Std column, because all pins are going to be in banks 34 and 35 we will pick the standard LVCMOS25 for 2.5V low voltage CMOS. This is because banks 34 and 35 reference voltage is set by the signal VADJ and the default for that value on the ZedBoard is 2.5V (jumper J10).

A fast way to make these settings across all 48 I/O pins is to use the GUI and the I/O Std column of the I/O Ports tab. Individual lines can be switched to LVCMOS25 or with the various GPIO entries collapsed

into a single line, the whole group can be set at once. Or the appropriate bank 34 and 35 lines can be copied from the Master XDC file.

Appendix A contains the full schematic for the KU HEP "L" Board which is the daughter card in this example. The pin assignments appear on page 2 along with the FMC connector schematic. The final two bits of GPIO\_B should be attached to SW0 and SW1 of the ZedBoard.

Once all the pins are mapped a bitstream can be created, exported and Vitis launched to create a new project.

Appendix B contains a sample XDC file reflecting all the connections needed to match the schematic in Appendix A.

### Manipulating Bits in GPIO IP

The definitive documentation for how the Xilinx AXI GPIO operates can be found in

[https://www.xilinx.com/support/documentation/ip\\_documentation/axi\\_gpio/v2\\_0/pg144-axi-gpio.pdf](https://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf)

From their Table 2-4 we learn there are 7 addresses in the processor memory space (all devices on an AXI bus are memory mapped). Because we are not using the interrupt options we only need to be

concerned with the first four entries in their table. Two registers for reading or writing data to the I/O pins and two registers for setting the data direction of the I/O pins.

To take advantage of the AXI GPIO IP we need a Vitis project. Start by modifying a “hello world” C project. Then we have the option of using the GPIO helper functions found in “xgpio.h” or directly manipulating the registers in the AXI GPIO. Depending on the speed of execution desired, either can be used or they can be intermixed.

It will be necessary to know the exact memory address assigned as the “base address” for the GPIO. This can be found in “xparameters.h” directly or indirectly using the XGpio\_Initialize() helper function.

Following this we need to set up the default data direction (DDR) by programming a 0 or 1 into each corresponding bit position for all of GPIO\_A and GPIO\_B. Programming a 0 makes the signal an output, programming a 1 makes it an input. In the case of lines that won’t be changing their data direction, such as LED control or reading switches, a good shortcut is to use the C-preprocessor to automatically compute the “magic” numbers.

```
#define INBIT (u32)1
#define OUTBIT (u32)0
#define DDR ((INBIT << 31) | \
              (INBIT << 30) | \
              (INBIT << 1) | \
              (INBIT << 0))
```

DDR will contain 0xC0000003 using this method. A further refinement would be to create #define’d bit positions such as

```
#define LED0 (u32)0
#define LED1 (u32)1
#define LED2 (u32)14
#define SW0 (u32)3
#define DDRLED ((OUTBIT << LED0) | \
                (OUTBIT << LED1) | \
                (OUTBIT << LED2) | \
                (INBIT << SW0))
```

Which makes the code somewhat self-documenting. It also means that if there is a design change and SW0 moves from bit-3 to bit-19, the only line that needs to be changed is the #define of SW0.

For DDR that need to change their state, using a variable to hold the current value and then manipulating the individual bits via AND/OR operations would be the most efficient. While it is possible to read back the contents of the DDR into a temporary register, manipulate bits and then re-write the DDR it is less efficient. Bit manipulation macros are easy to find using simple Google searches. But a better option is to create static inline functions and pass by reference the local copy of the register contents. For example:

```
static inline void set_bit(u32 *x, int bitNum) {
    *x |= (1L << bitNum);
}
```

There are helper functions defined in xgpio.h for also performing single and multiple bit set and clear operations in the various GPIO registers. These can be used directly or adapted to faster operation by removing some of the assert() function calls and using pre-computed register addresses.

To configure the data direction, the magic word such as “DDRLED” defined above needs to be written into a register of the GPIO. As we have defined a 2-channel GPIO, there will be two such DDR words. The address offset for these registers is 0x0004 (channel 1) and 0x000C (channel 2). If using the helper

functions in “xgpio.h” the functions of interest is “XGpio\_SetDataDirection()” for which the argument list includes the channel (1 or 2) and the DDR mask as well as the filled out Xgpio object from the initial call to XGpio\_Initialize(). Alternately, one can write directly to the base address of the GPIO module and compute the register offsets manually.

When writing or reading data from a GPIO, there are again two register offsets in a 2 channel GPIO module such as implemented in this project. The register offset for channel 1 data is 0x0000 (same as the base address) and for the 2<sup>nd</sup> channel the offset is 0x0008.

Since a bank of GPIO can be mixed input and output but only a single register exists, the writing of a port that was defined as input is ignored. And the reading of a port defined as output returns the last value written to the port.

There are functions defined in “xil\_io.h” that show how more direct I/O access to the GPIO is accomplished. These offer little to no protection from accessing memory outside the memory map of the design and so being faster can also cause odd behavior over the more protected “XGpio\_DiscreteRead()” and “XGpio\_DiscreteWrite()” function calls.

Finally, it is best not to allow output bits to “twinkle”, that is to say if there are many outputs that must change in a port don’t change them individually, updating the port on each bit change. If possible change all bits simultaneously. If there are timing requirements such as a break-before-make condition or an edge transition that must precede a read of a data bit, these bits could be manipulated individually. To this end, keeping a shadow copy of the register in a memory variable allows for updating the entire register simultaneously.

For example:

```
u32 shadow_A;
shadow_A = (u32)0b10101010;
Xil_Out32(GPIO_A, shadow_A);
/* do something related to the output bits */
Shadow_A = shadow_a ^ (u32)0xff; // flip the lower 8 bits
Xil_Out32(GPIO_A, shadow_A);
```

When reading multiple bits, the use of shift and mask operators can isolate a single bit from the register.

```
U32 gpio_b_read;
Gpio_b_read = Xil_in32(GPIO_B) & SW_MASK;
Sw0 = (gpio_b_read >> SW0_BIT_POSITION) & 1; // isolate switch state as 0 or 1
```

Here SW\_MASK and SW0\_BIT\_POSITION were defined previously. If they are done using #define it may produce faster executable code as the C pre-processor and C compiler will recognize them as constants. Thus generating instructions using direct memory addressing instead of a more complicated and slower indirect addressing model.

## Debugging GPIO I/O

Because the Zync (and Microblaze) processors have all peripherals as memory mapped I/O the debugger can be used to watch for changes in peripheral registers. With the limitation that if code execution is paused due to a breakpoint, the I/O may change state (somebody flips a switch from 0 to 1 on a bit of a GPIO register for example) without the debugger being aware until the next execution cycle of the processor. Breakpoints can be set in Vitis based on change of a memory address which can be helpful in debugging code that is heavy in I/O interactions.

### Final Exercise

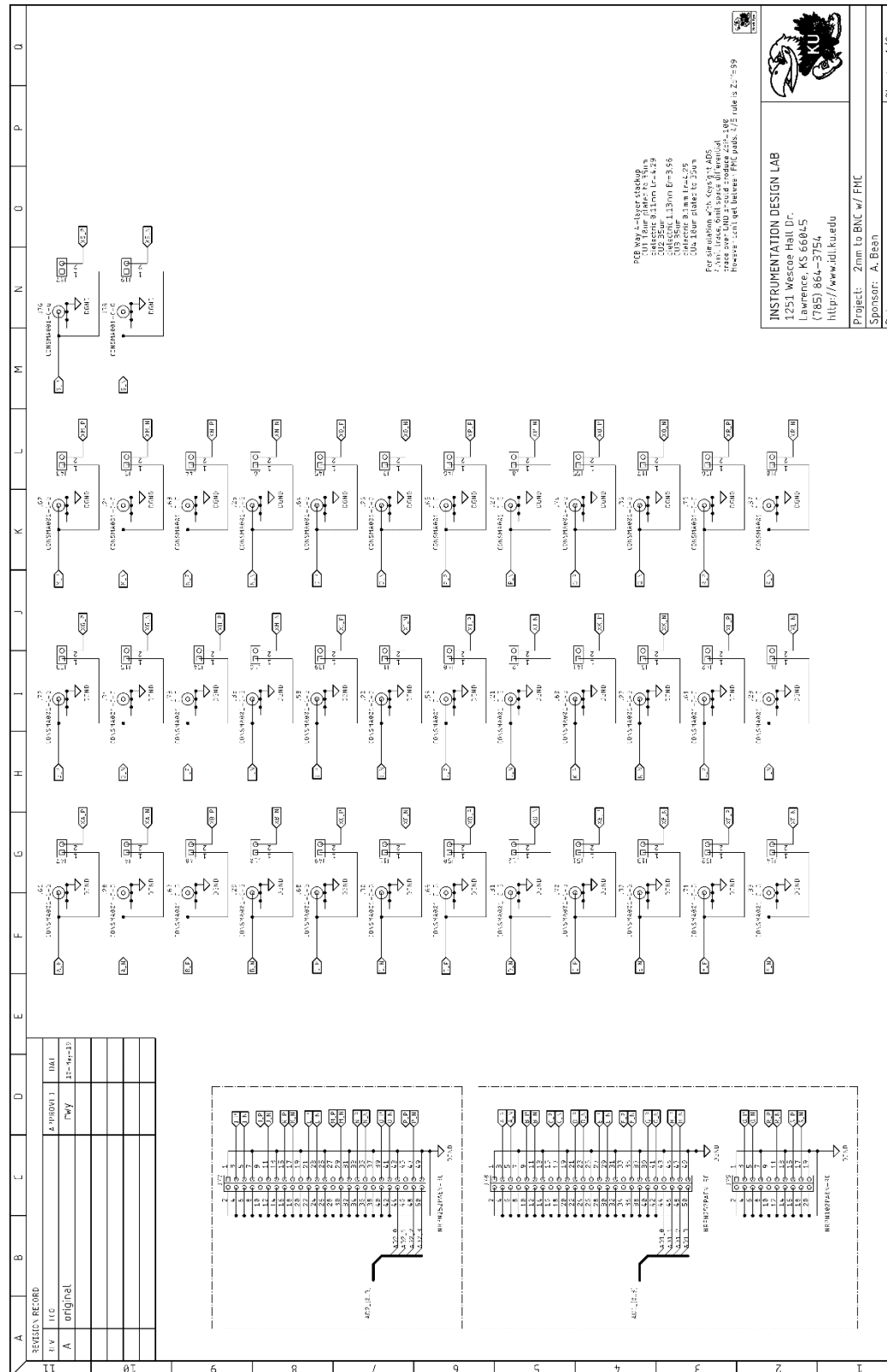
Because the I/O of the GPIO ports has been mapped to the FMC connector it is somewhat difficult to access. But two additional signals of GPIO\_B can be mapped to the two switches SW0 and SW1 of the zedboard.

By mapping GPIO\_B[14] to SW0 and GPIO\_B[15] to SW1, write a program that will monitor and display the values of the switches (0 or 1) on a terminal such as Tera Term. Consider methods for minimizing the flicker of the terminal screen such as limiting the time spend displaying updates to a minimum or only displaying an update if a change is detected.

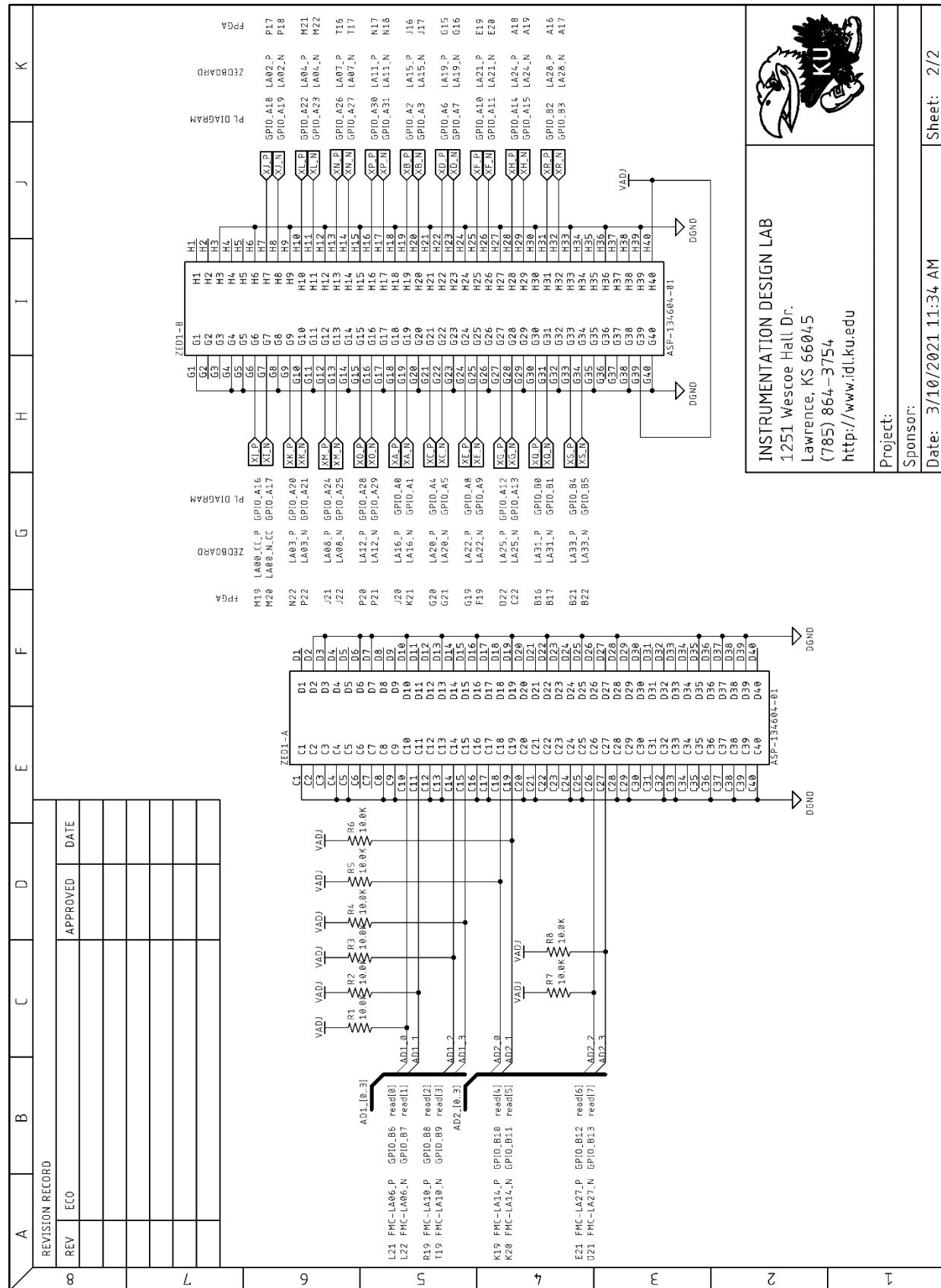
A potential solution appears in Appendix C.



## Appendix A – KU HEP “L” Board Schematic







## Appendix B – Sample XDC File

```

set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[31]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[30]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[29]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[28]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[27]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[26]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[25]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[24]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[23]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[22]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[21]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[20]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[19]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[18]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[17]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[16]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[15]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[14]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[13]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[12]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[11]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[10]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[9]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[8]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[7]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[6]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[5]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[4]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[3]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[2]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[1]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_A_tri_io[0]}]

set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_B_tri_io[15]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_B_tri_io[14]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_B_tri_io[13]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_B_tri_io[12]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_B_tri_io[11]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_B_tri_io[10]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_B_tri_io[9]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_B_tri_io[8]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_B_tri_io[7]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_B_tri_io[6]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_B_tri_io[5]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_B_tri_io[4]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_B_tri_io[3]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_B_tri_io[2]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_B_tri_io[1]}]
set_property IOSTANDARD LVCMOS25 [get_ports {GPIO_B_tri_io[0]}]

set_property PACKAGE_PIN J20 [get_ports {GPIO_A_tri_io[0]}];
set_property PACKAGE_PIN K21 [get_ports {GPIO_A_tri_io[1]}];

set_property PACKAGE_PIN J16 [get_ports {GPIO_A_tri_io[2]}];
set_property PACKAGE_PIN J17 [get_ports {GPIO_A_tri_io[3]}];

set_property PACKAGE_PIN G20 [get_ports {GPIO_A_tri_io[4]}];
set_property PACKAGE_PIN G21 [get_ports {GPIO_A_tri_io[5]}];

set_property PACKAGE_PIN G15 [get_ports {GPIO_A_tri_io[6]}];
set_property PACKAGE_PIN G16 [get_ports {GPIO_A_tri_io[7]}];

set_property PACKAGE_PIN G19 [get_ports {GPIO_A_tri_io[8]}];
set_property PACKAGE_PIN F19 [get_ports {GPIO_A_tri_io[9]}];

set_property PACKAGE_PIN E19 [get_ports {GPIO_A_tri_io[10]}];
set_property PACKAGE_PIN E20 [get_ports {GPIO_A_tri_io[11]}];

set_property PACKAGE_PIN D22 [get_ports {GPIO_A_tri_io[12]}];

```

```
set_property PACKAGE_PIN C22 [get_ports {GPIO_A_tri_io[13]}};

set_property PACKAGE_PIN A18 [get_ports {GPIO_A_tri_io[14]}};
set_property PACKAGE_PIN A19 [get_ports {GPIO_A_tri_io[15]}};

set_property PACKAGE_PIN M19 [get_ports {GPIO_A_tri_io[16]}};
set_property PACKAGE_PIN M20 [get_ports {GPIO_A_tri_io[17]}};

set_property PACKAGE_PIN P17 [get_ports {GPIO_A_tri_io[18]}};
set_property PACKAGE_PIN P18 [get_ports {GPIO_A_tri_io[19]}};

set_property PACKAGE_PIN N22 [get_ports {GPIO_A_tri_io[20]}};
set_property PACKAGE_PIN P22 [get_ports {GPIO_A_tri_io[21]}};

set_property PACKAGE_PIN M21 [get_ports {GPIO_A_tri_io[22]}};
set_property PACKAGE_PIN M22 [get_ports {GPIO_A_tri_io[23]}};

set_property PACKAGE_PIN J21 [get_ports {GPIO_A_tri_io[24]}};
set_property PACKAGE_PIN J22 [get_ports {GPIO_A_tri_io[25]}};

set_property PACKAGE_PIN T16 [get_ports {GPIO_A_tri_io[26]}};
set_property PACKAGE_PIN T17 [get_ports {GPIO_A_tri_io[27]}};

set_property PACKAGE_PIN P20 [get_ports {GPIO_A_tri_io[28]}};
set_property PACKAGE_PIN P21 [get_ports {GPIO_A_tri_io[29]}};

set_property PACKAGE_PIN N17 [get_ports {GPIO_A_tri_io[30]}};
set_property PACKAGE_PIN N18 [get_ports {GPIO_A_tri_io[31]}};

set_property PACKAGE_PIN B16 [get_ports {GPIO_B_tri_io[0]}};
set_property PACKAGE_PIN B17 [get_ports {GPIO_B_tri_io[1]}};

set_property PACKAGE_PIN A16 [get_ports {GPIO_B_tri_io[2]}};
set_property PACKAGE_PIN A17 [get_ports {GPIO_B_tri_io[3]}};

set_property PACKAGE_PIN B21 [get_ports {GPIO_B_tri_io[4]}};
set_property PACKAGE_PIN B22 [get_ports {GPIO_B_tri_io[5]}};

set_property PACKAGE_PIN L21 [get_ports {GPIO_B_tri_io[6]}};
set_property PACKAGE_PIN L22 [get_ports {GPIO_B_tri_io[7]}};

set_property PACKAGE_PIN R19 [get_ports {GPIO_B_tri_io[8]}};
set_property PACKAGE_PIN T19 [get_ports {GPIO_B_tri_io[9]}};

set_property PACKAGE_PIN K19 [get_ports {GPIO_B_tri_io[10]}};
set_property PACKAGE_PIN K20 [get_ports {GPIO_B_tri_io[11]}};

set_property PACKAGE_PIN E21 [get_ports {GPIO_B_tri_io[12]}};
set_property PACKAGE_PIN D21 [get_ports {GPIO_B_tri_io[13]}};

set_property PACKAGE_PIN F22 [get_ports {GPIO_B_tri_io[14]}};
set_property PACKAGE_PIN G22 [get_ports {GPIO_B_tri_io[15]}};
```

## Appendix C – Read Switches

```

Gpio_mapping.h:
#ifndef __gpio_mapping_h__

#include "stdio.h"
#include "sleep.h"
#include "platform.h"
#include "xil_printf.h"
#include "xgpio.h"
#include "xuartps_hw.h"

#define INBIT (u32)1
#define OUTBIT (u32)0
#define COWSNOTHOME 1

/* http://braun-home.net/michael/info/misc/VT100_commands.htm */
#define VT100_ESC 27 // Tera Term emulates VT100, define the escape code
#define ClearScreen xil_printf("%c[2J",VT100_ESC) // clear screen using VT100 format string
#define CursorHome xil_printf("%c[f",VT100_ESC) // cursor home
#define BLINK xil_printf("%c[5m",VT100_ESC); // enable blink
#define NORMAL xil_printf("%c[0m", VT100_ESC); // normal printing

#define UART_BASEADDR XPAR_XUARTPS_0_BASEADDR

#define GPIO0 XPAR_GPIO_0_DEVICE_ID
#define LED XPAR_GPIO_0_BASEADDR
#define LED_CHANNEL 1
#define GPIO1 XPAR_GPIO_1_DEVICE_ID
#define GPIO_A XPAR_GPIO_1_BASEADDR + XGPIO_DATA_OFFSET
#define GPIO_B XPAR_GPIO_1_BASEADDR + XGPIO_DATA2_OFFSET
#define A_CHANNEL 1
#define B_CHANNEL 2
#define SW0 14 // bit position in GPIO_B
#define SW1 15 // bit position in GPIO_B
#define SW_MASK (u32)((1 << SW0) | (1 << SW1))
#define A_DEFAULT_DDR 0xffffffff
#define B_DEFAULT_DDR 0xffffffff // only lower 16 bits count

/* helper functions for bit set and clear in a field */
static inline void set_bit(u32 *x, int bitNum) {
    *x |= (1L << bitNum);
}

static inline void clear_bit(u32 *x, int bitNum) {
    *x &= ~(1L << bitNum);
}

XGpio Gpio_LED;
XGpio Gpio_IO;
volatile u32 shadow_led;
volatile u32 shadow_A; // for writing
volatile u16 shadow_B; // for writing

int InitGPIO(void);

#endif // __gpio_mapping_h__

Gpio_mapping.c

#include "gpio_mapping.h"

int main() {
    int x=0;
    u32 gpio_b_read;
    int sw0, sw1, sw0old, sw1old;

```

```

init_platform();

ClearScreen;
CursorHome;

xil_printf("Hello world!\r\n");

if (InitGPIO() != XST_SUCCESS)
    xil_printf("\r\nError initializing GPIO");

xil_printf("\r\n\n");
sw0old = -99;
sw1old = -99;
xil_printf("changes\tsw1\tsw0\r\n");
while(COWSNOTHOME)
{
    gpio_b_read = Xil_In32(GPIO_B) & SW_MASK;
    sw0 = (gpio_b_read >> SW0) & 1;
    sw1 = (gpio_b_read >> SW1);

    if ((sw0old != sw0) || (sw1old != sw1))
    {
        BLINK; // do obnoxious things to display
        xil_printf("%d", x++);
        NORMAL;
        xil_printf("\t%d\t%d \r", sw1, sw0);
        sw0old = sw0;
        sw1old = sw1;
        usleep(20000); // cheap and dirty debounce by wasting 20ms
    }
}

cleanup_platform();
return 0;
}

int InitGPIO(void) {
    int Status;

    // use helper functions to get base address of GPIO
    Status = XGpio_Initialize(&Gpio_LED, GPIO0);
    if (Status != XST_SUCCESS) {
        xil_printf("\r\nGPIO init failed GPIO0");
        return (XST_FAILURE);
    }
    Status = XGpio_Initialize(&Gpio_IO, GPIO1);
    if (Status != XST_SUCCESS) {
        xil_printf("\r\nGPIO init failed GPIO1");
        return (XST_FAILURE);
    }

    // use helper functions to set data direction bits
    XGpio_SetDataDirection(&Gpio_IO, A_CHANNEL, A_DEFAULT_DDR);
    XGpio_SetDataDirection(&Gpio_IO, B_CHANNEL, B_DEFAULT_DDR);

    // directly write to data address to set bits
    shadow_A = (u32)0;
    shadow_B = (u32)0;
    Xil_Out32(GPIO_A, shadow_A);
    Xil_Out32(GPIO_B, shadow_B);

    shadow_led = (u32)0;
    Xil_Out32(LED, shadow_led);

    return (XST_SUCCESS);
}

```