ECE527

MP1 Report

Tan Huu Nguyen (Net ID: thnguyn2)
Enyu Luo (Net ID: enyuluo1)

Part A - Basic I/O

1. Requirements

Implement a system that reads states of the GPIO slide switches on the board and displays them on the onboard LEDs. Any changes on the input (switches), should be reflected at the ouput (leds) after exactly 3 clock cycles.

2. Assumptions

Since the status of the output is not specified by the design requirement, we choose the following assumption. When reset button is pressed, all the outputs will be OFF.

· After the reset button is released, any change in the inputs will be reflected to the output after 3 clock cycles.

3. Functional block diagram

We used 8 instances of a 3-bit shift register for 8 input-output pairs. Shown below is a functional block diagram for 1 pair. D-Flip Flops (FFs) are used in these shift registers. A single clock signal (CLK) and a single reset signal (RESET) are used for all the FFs. Since the LEDs are connected to the Q port of the FFs. They are buffered.

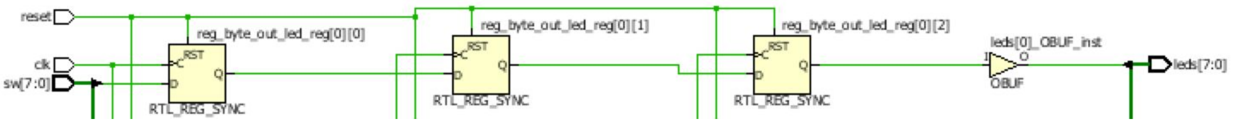


Figure 1: Block Diagram of Implementation (Partial Schematic)

Figure 1 shows our implementation of the 3-bit shift register which causes the 3 cycle delay. Since a single flip flop contributes to a single cycle delay, 3 cycle delay is achieved by cascading 3 flip-flops. This design is implemented in parallel with all 8 bits with switches as inputs and outputs to LEDs.

4. Entities / Modules

There is only one module, called partA. This module contains all eight 3-bit shift registers that implement the 3-cycle delay. This module is also the top-level module. It has a single input for clock, single input for reset, 8 bit input for switches and 8-bit outputs for LEDs. A parameter name DELAY_NUM_CYCLES determines how many clock cycles will be delayed. Here, this variable is 3. In this module, we declare an 8-element array of 3-bit shift register. For each clock cycle, the most significant bit of each register will be assigned to the LED output and all remaining bits will be combined with the input value of the corresponding input switch before writing back to the shift register.

5. *Design process*

To make sure that the delay is exactly 3 clock cycles, we use sequential logic with D-FFs in which we know exactly how many cycles are needed for an input bit to reach the output. Since 1 clock cycle will shift the register by 1 bit. We also look at the elaborated and synthesized schematic to double-check whether the design goals are met or not.

6. *Post-implementation results*

- Resource Utilization

Resource	Utilization	Available	Utilization %
FF	24	106400	0.02
I/O	18	200	9.0
BUFG	1	32	3.12

- Power

Total On-Chip Power: 0.128 W
Junction Temperature: 26.5ºC
Thermal Margin: 58.5 ºC (4.9 W)
Effective djA: 11.5ºC/W
Power supplied to off-chip device: 0 W
Confidence level: Low

- Timing

Worst Negative Slack (WNS): 8.666 ns
Total Negative Slack (TNS): 0 ns
Number of Failing EndPoint: 0
Total Number of Endpoints: 16

7. *Difficulties/bugs encountered*

It takes us a lot of time to get familiar with the board, get it working as well as linking the license to Vivado. However, thanks to the help from the TA, things are all set.

8. *What we learnt from the problem*

We got familiar with the Vivado environment, choosing board, assigning I/Os, synthesizing the project, generating bitstream and programming bitstream into the FPGA.

9. *Vivado version and computer usage*

We use Vivado v2015.1. Enyu Luo is using the software on a private Mac running Windows 8 VM. Tan Nguyen is working on Ubuntu Linux.

Part B - Basic IO and FSM

1. Requirements

Implement a system that reads the GPIO switch states and displays them on the onboard LEDs. The system will operate in four modes, triggered by the onboard push buttons.

· (Button Up) Mode 0: Display states of switches
· (Button Down) Mode 1: Display state of switches , logically right shifted by 2
· (Button Left) Mode 2: Display state of switches, circular shifted to the left by 3
· (Button Right) Mode 3: Display states of switches, logically inverted
· (Button Center) Mode Reset

2. Assumptions

Since the problem statements state that the modes are triggered by onboard push buttons, and did not specify a default mode for reset state, we assumed that reset causes all LED outputs to be '0' and does not operate in any of the 4 modes till a mode button is pressed. We also assume that after the button is released, the state stays the same until another button is pressed. The mode is changed only when a level '1' is detected in one of 5 inputs for 5 buttons using combinatorial logic. Therefore, we don't need to do button debouncing.
When two or more buttons are pressed at the same time, only one mode is selected. That is the mode of highest priority. The priority is listed in the following decreasing order:
Up button (Mode 0)> Down button (Mode 1) > Left button (Mode 2) > Right button (Mode 3)> Center button (ModeRST).
Also, we enforce the system to go into reset mode right after the system is booted up and before any key is pressed.

3. Functional block diagram

Figure # below shows the block diagram of the whole system. Upon pressing one of the push buttons, the state machine will change state at the next clock cycle. Combination logic, implemented as lookup tables in the FPGA is used to decide what to output based on the current state, and the input slide switches.

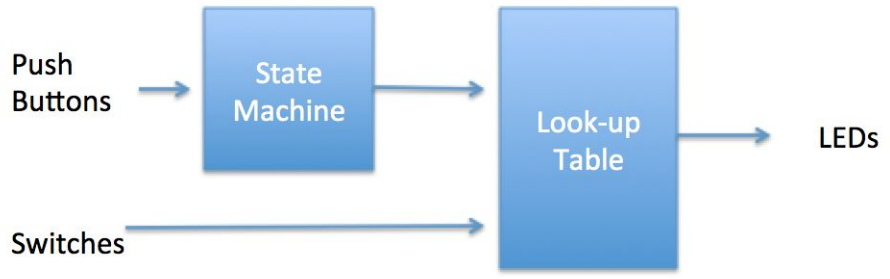


Figure : Part B Block Diagram

4. *Entities / Modules*

Only 1 module is used, named part_B. Inputs are clk, reset(center button), 4 other buttons, and the 8 slides switches. Outputs are the 8 LEDs. This module implements both the state machine and the combinational logic for the system to work. The module consist of following code blocks:

- Enforce the current state and the next state variable into the reset mode in the beginning at the rising edge of the clock signal.
- Defining the next step logic using sequential logic at each rising edge of the clk signal
- Using combinatorial logic to define the next state logic when one of the button is pressed.
- Define the output logics for each different state.

5. *Design process*

We had two parts to the design, one part is the state machine controller, and the other part is the logic for producing the correct outputs in the various modes.

For the state machine controller, we defined 5 states. 4 states correspond to each mode as specified by the requirement. The fifth state is defined as the reset state where all outputs will be zero until a certain mode push button is pressed. Upon a button push, the next state variable is updated, which gets 'flip-flopped' into the current state at the next clock cycle. This is verified by looking at the schematic generated.

A case statement is used to implement the logic for the LED outputs. Behavior is defined according to the current state of the state machine. This could be the same values as input slide switches, the shifted version or the flipped version. This is compiled into a lookup table seen in the schematic design.

6. *Post-implementation results*

- Resource Utilization

Resource	Utilization	Available	Utilization %
FF	3	106400	0.01
I/O	22	200	11.0
BUFG	1	32	3.12
LUT	11	53200	0.02

- Power

Total On-Chip Power: 0.134 W
Junction Temperature: 26.5ºC
Thermal Margin: 58.5 ºC (4.9 W)
Effective djA: 11.5ºC/W
Power supplied to off-chip device: 0 W
Confidence level: Low

- Timing

Worst Negative Slack (WNS): 8.746 ns
Total Negative Slack (TNS): 0 ns
Number of Failing EndPoint: 0
Total Number of Endpoints: 3

7. Difficulties/bugs encountered

We did not experience any difficulties in this part.

8. What we learnt from the problem

This part is basically a slightly more complex design compared to part A. It's a simple practice for state machine design and learnt that combination logic is implemented as lookup tables in FPGAs.

9. Vivado version and computer usage

We use Vivado v2015.1. Enyu Luo is using the software on a private Mac running Windows 8 VM. Tan Nguyen is working on Ubuntu Linux.

Part C - Embedded SoC

1. Requirements

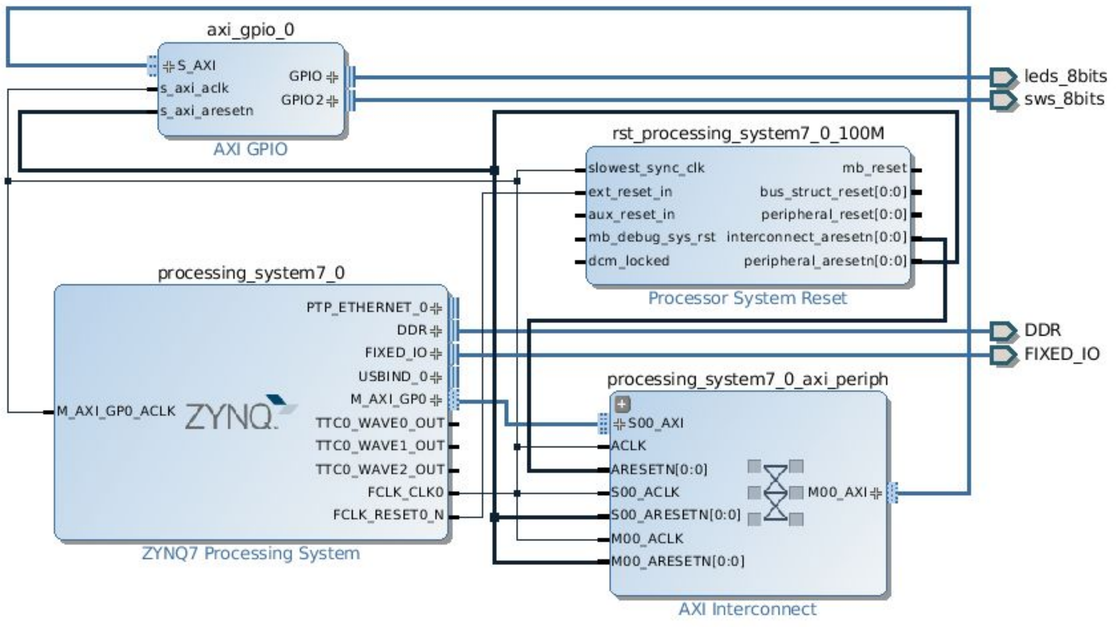
Implement a Zynq based SoC system to produce a similar output as in Part B, by interfacing the PS and PL sides via an AXI interface. The program will

1. Connect the LEDs and the Switches via AXI GPIO.
2. Read the switch positions via AXI
3. Write to the LEDs via AXI
4. Cycle through the 4 modes described in Part B, with 1 second latencies.

2. Assumptions

Since push buttons are not required, we assume that the center button is also not needed to function as a reset button, thus the system only cycles through the 4 modes without a reset feature.
We also assumed that the LEDs do not have to change in the middle of the 1 second duration, and they only get updated at the start of the one second duration for each mode cycle.

3. *Functional block diagram*



The design uses the ZYNQ processing module, connected to the LEDs and switches on the Programming Logic using the AXI_GPIO. An AXI Interconnect block is used to connect these two IP instances. Reset signals are provided by the Processor System Reset module. The software for the ZynQ processor can be explained by the following pseudo-code

```
//------------------------------------------------------------------------------------------------------------------
Initialize_AXI_GPIO();
repeat
    begin
        execute_Mode_0(); //Read input switches and display to LEDs
        delay_1s();
        execute_Mode_1();//Read input switches, right shift by 2 bits and display
        delay_1s();
        execute_Mode_2();//Read switches, circularly shift to the left by 3 bits and disp.
        delay_1s();
        execute_Mode_3();//Read switches, flip all the bits and display
        delay_1s();
    end
//------------------------------------------------------------------------------------------------------------------
```

4. *Entities / Modules*

The hardware part consists of the processing system, Processor System Reset, AXI Interconnect, and AXI GPIO. These entities are provided by Xilinx to allow the software part to interact with the gpio, particularly the switches and LEDs as required for this mp.

The software part consists of a main super loop. This loop ensures that our code keeps running and not return to a location in program memory that has undefined behavior. This loop also cycles between each of the 4 modes, and contains the behavior of the LED outputs in written in C. The 1 second delay is implemented by a for loop.

5. *Design process*

To ensure an approximate value of 1s for the delay is produced, we used a For loop with a range value for the loop variable specified by time measurements. First, a very large value for the loop variable is used. With this value, we measured the delay generated using an iPhone in stopwatch mode. In our experiment, a delay of 9.51 s is measured. Then, we scaled down the value of the loop variable by a factor of 9.51 and then, rounded the value to the closest integer. Finally, we choose 88,888,888 as our range of the loop variable.

6. *Post-implementation results*

- Resource Utilization

Resource	Utilization	Available	Utilization %
FF	668	106400	0.63
LUT	484	53200	0.91
Memory LUT	64	17400	0.37
I/O	16	200	8.00
BUFG	1	32	3.12

- Power

Total On-Chip Power: 1.697 W
Junction Temperature: 44.6ºC
Thermal Margin: 40.4 ºC (3.4 W)
Effective djA: 11.5ºC/W
Power supplied to off-chip device: 0 W
Confidence level: Low

- Timing

Worst Negative Slack (WNS): 4.154 ns
Total Negative Slack (TNS): 0 ns
Number of Failing EndPoint: 0
Total Number of Endpoints: 1395

7. *Difficulties/bugs encountered*

We did not get much difficulties in this part except we must use a separate timer to measured the scaled-up delay time and then, scaled down to get close to 1s delay. However, this is a very rough way to do it and 1s delay is not obtained at high accuracy.

8. What we learnt from the problem

We learned how to do hardware-software codesign, how to use the block designer to implement the ZynQ based SOC system, how to connect to PL's IOs using AXI_GPIO.

9. Anything extra you feel like to add

For a more accurate 1 second delay, we could have added a hardware timer that causes an interrupt to the processing system. Or, we could run linux that uses the system tick timer and use POSIX libraries to get a 1 second delay.

10. Vivado version and computer usage

We use Vivado v2015.1. Enyu Luo is using the software on a private Mac running Windows 8 VM. Tan Nguyen is working on Ubuntu Linux.