

Lab 4: Design of MicroBlaze MCS based System

Chaiwat Ekkaewnumchai

ECE Box #693

Mar 2, 2018

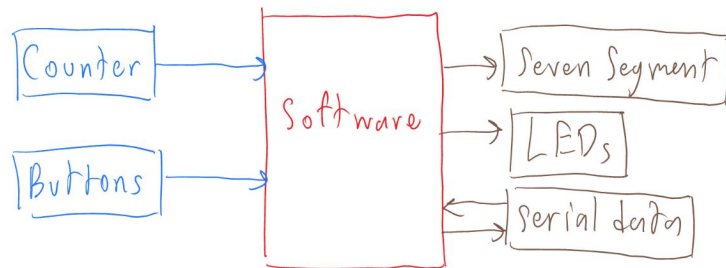
Introduction

This lab involves on-board Microblaze Microcontroller System (MCS), which has its own methodology to take inputs and outputs in serial-form data. Additionally, the lab also requires user to be familiar with the style Xilinx provides because there is a complete implementation to connect between FPGA board and microcontroller. This lab also needs a good design to cope with the troubles encountered on previous labs such as button debouncing.

Discussion

Implementation

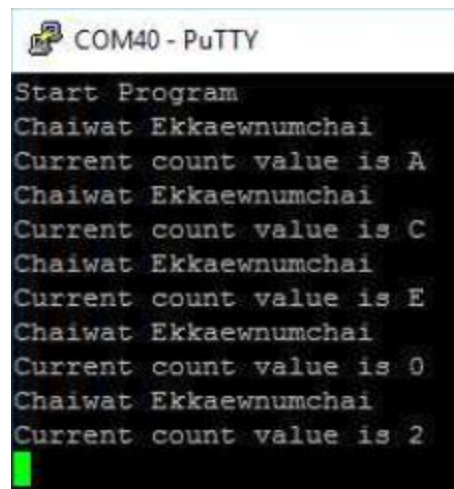
The diagram roughly shows that the module takes inputs from buttons and hardware counter (and sometime serial data) to software part. Then, the software part will send data back to seven segment, LEDs, and serial data to PC according to button inputs.



First, the hardware part has to be wired. Seven segment and counter modules using clock are familiar to this course and not too complicated. Other than that, LEDs and buttons are also wired to the hardware part. In addition, button debouncing should be handled properly. Similarly to lab 3, simple state machine is added per one button to delay the pressing consideration; state machine has 4 states to indicate the length of time that a button is pressed. If it is pressed long enough, the module signal via a wire that a button is pressed or unpressed. Unlike lab 3 that one can assume that the change can be happen completely in one clock edge cycle, there is another input signal for the module to ensure that the needed changes have really gone to the software part; the software part will input it to 1 if it takes the button indicator already. More detail is described in the next paragraph.

Next, Microblaze MCS module is created; some information are required to be specified. Because the software part should be able to receive button and counter signals and send back signals to control three of four panels on seven segment and sixteen LEDs, 8 and 28 bits of General Purpose Input (GPI) and General Purpose Output (GPO) are needed. GPO is separated to 2 channels: LEDs channel (channel 1) and seven segment channel (channel 2). In addition, more 4 bits GPO are also sent back in order to signal the hardware that button pressing is already read from software part.

To implement the software part, one needs to know that what information is retrieved from which input channel and what information should be sent back through which output channel. In this case, it is designed that the first four bits are buttons signal after button debouncing handler and the last four bits are the counter value (from 0 to F). For output channels, data are separated to three channels: LEDs, seven segment values, and button feedback. Each bit of LED channel represents each LED; the most significant bit is for the left-most LED on the board, while the least significant bit is for the right-most LED on the board so that one can translate the bits easier. Hence, to set the pattern described in the project description, the right shift register is used as well as the or-operation to assign the most significant bit to 1; 1 should be put to the result and move all set bit by one bit, which means exactly the same functionality in the project description. Next is the seven segment value. An input character that is typed in a serial port is in ASCII whose number character and alphabet character do not have the same values as in hexadecimal form. Hence, the input value should be changed before used; because 0 to 9 has consecutive value as well as A to F, the values can be subtracted to 0 or A to get directly usable value for number and alphabet respectively. In order to ensure that all three characters can be get correctly, one character is read at a time for three times. Because all operations can be carried out in any order at any time, the latest values to print out on seven segment and the pattern of LEDs lit are kept in software variables so that for every loop they can be sent out correctly. Moreover, data has to be sent to the hardware part as described earlier. Two last objectives can be achieved similarly. One is to print out static string that is our name, while other is to print out a retrieved value. Values can be retrieved in similar fashion to sending out. There is a function that can be called to print out string similar to printf in C, which we are using for printing our name and counter value. Because the value for counter is in hexadecimal form when retrieved directly from the last four bits of the input channel, we can print it out in hexadecimal value directly. The picture on right side shows the output from serial port COM40, which received the data from Microblaze MCS board. The results were from pressing the first and second instruction buttons that printed my name and current value of hardware counter respectively. Unfortunately, the remaining instruction



```
COM40 - PuTTY
Start Program
Chaiwat Ekkaewnumchai
Current count value is A
Chaiwat Ekkaewnumchai
Current count value is C
Chaiwat Ekkaewnumchai
Current count value is E
Chaiwat Ekkaewnumchai
Current count value is 0
Chaiwat Ekkaewnumchai
Current count value is 2
```

cannot be printed out to standard output. Also, an input from standard input (keyboard) is also not printed out.

FPGA Resource Usage

Utilization			
		Post-Synthesis	Post-Implementation
Graph Table			
Resource	Estimation	Available	Utilization %
LUT	275	20800	1.32
FF	206	41600	0.50
IO	34	106	32.08

Utilization			
		Post-Synthesis	Post-Implementation
Graph Table			
Resource	Utilization	Available	Utilization %
LUT	828	20800	3.98
LUTRAM	175	9600	1.82
FF	565	41600	1.36
BRAM	8	50	16.00
IO	35	106	33.02
BUFG	2	32	6.25
MMCM	1	5	20.00

The flip flop counting process iterates in `mcs_top.v` declaration order.

In `mcs_top.v`, 27 flip flops are required to count from 0 to 99,999,999 to divide clock from 100MHz to 1Hz. 4 flip flops are required to implement hardware counter to count from 0 to F. There are more flip flops inherited in `button_enable` and `seven_seg` modules.

In `button_enable.v`, 2 flip flops are required to slow down clock (from 100MHz). there are 4 buttons in total. 2 flip flops are required to keep track a state for each button. There are more flip flops inherited in `button_trigger` module, which is instantiated 8 times.

In `button_trigger.v`, 17 flip flops are required for the counter of clock divider to count from 0 to 119999. 2 flip flops are required for The state of the state machine to indicate how many cycle passed. The total of the module is 19 flip flops.

In `seven_seg.v`, 11 flip flops are required for the counter of clock divider to count from 0 to 2000 to delay too high frequency clock for seven segment. 2 flip flops are required for the selection of seven segment panel to easier handle the position on seven segment.

Therefore, in `mcs_top` module, there are 31, 162, and 13 flip flops from `mcs_top` itself, `button_enable`, and `seven_seg`. That is, there are 206 flip flops in total, which matches the number of resource on post-synthesis.

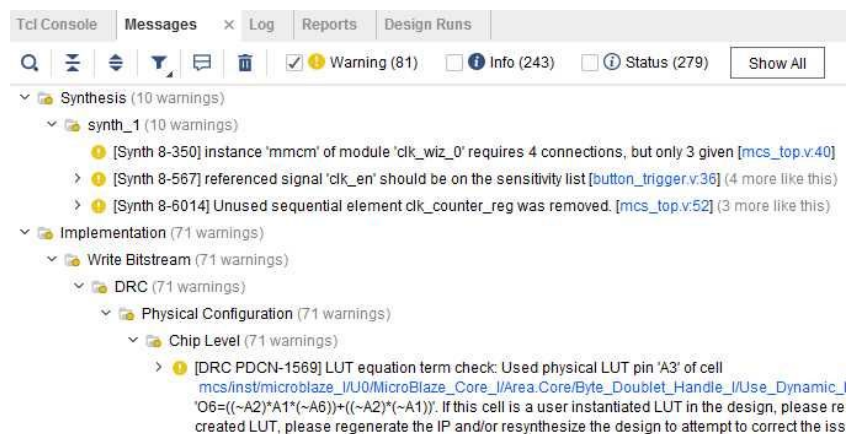
Problems Encountered

The first problem happened when we started to test LEDs lighting from the most left to the most right. However, only the middle to the most right were lit. This problem was from unconsciously misunderstanding about the hexadecimal form, which we were somehow confused that in ASCII form one letter represents in 8 bits while in hexadecimal one letter represents in 4 bits. Then, it became simple; adding more two zero in hexadecimal is enough.

The last and biggest problem we encounter in this lab is button reading with debouncing handler. In previous lab, the buttons were used directly to assign some values on hardware.

However, in this lab, a signal should be sent to the software part before it is set back to already-used state. Initially, we stuck on the debouncing problem and tried to solve it the same way we did for previous lab. However, the software part could not receive button signal regardless of how long and hard we pushed. Fortunately, we noticed that the button was once set to 1 but not long enough for the software to read because the mechanism that the hardware sent to the software; it sends one bit at a rising clock edge, where the signal was already reset to 0. The solution is adding one more signal per button to indicate that the button is already read by the software; this appears in `button_enable` parameters input called `*_signal`.

Warning



The first warning is about missing an argument on Mixed-Mode Clock Manager (MMCM), which is not used in the top module. The second warning is about using a variable to decide assignments; `clk_en` is used to indicate any change made, which will change value on the rising edge of a main clock. The last warning on synthesis section is that some variables are not used directly but indicate states. The remaining warning from implementation section can be ignored according to the instruction document.

Conclusion

We can accomplish in designing and implementing circuits to interact with embedded microcontroller. Precisely, we learn how to use pre-implemented module in IP Catalog similarly to how to create MMCM. Also, we learn how hardware part and software part can communicate. As expected, many problems encountered while working on this lab. One of them is button denouncing with knowing that the button may not be used at the short period of time similarly to the previous lab. Adding more signal to be able to tell hardware that the changes are applied and thus the states can be altered after that is a good solution. In conclusion, we can

successfully design electronic circuits to transfer data back and forth to microcontroller to achieve the goals. In the future, we can add more functionality by using slide switch or combination of buttons.

Appendix