Introduction

In this lab, a Vivado project file that includes a MicroBlaze soft processor was provided as the starting point to create an application that performs the following functions: read the push buttons and dip switches on the Basys3 board, set the LED and 7 segment display outputs, create a frequency controlled square wave output to the AMP2 Audio Amplifier to generated different musical notes, and generate a short tune on start up.

Solution Overview

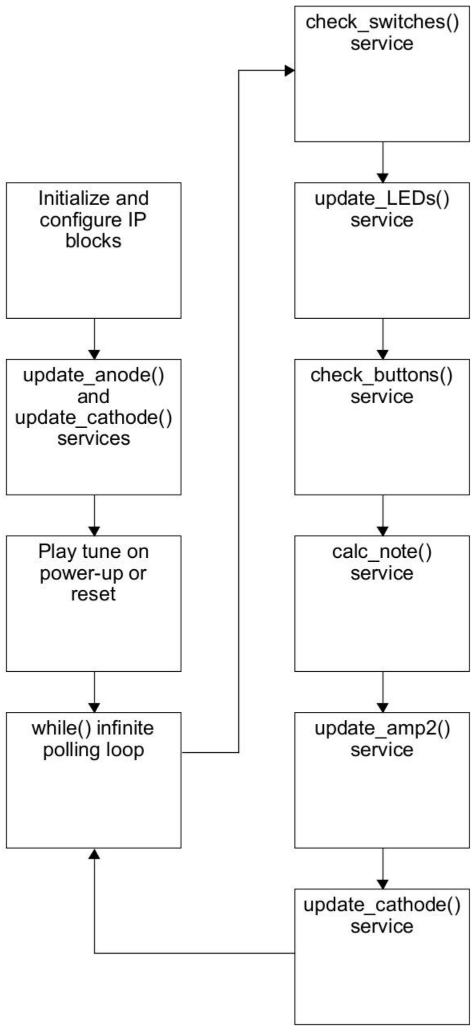
Initialize and configure IP blocks
update_anode() and update_cathode() services
Play tune on power-up or reset
while() infinite polling loop
check_switches() service
update_LEDs() service
check_buttons() service
calc_note() service
update_amp2() service
update_cathode() service

Figure 1: Flow diagram of lab

After initializing and configuring the required IP blocks, when the Basys3 board powers up or after a reset, it plays a wake up tune with 5 notes, each long enough to be heard. Then, the values of the buttons and slider switches are read and change the toggle rate of the AMP2's Analog Input to create a musical note corresponding to the settings defined in the provided table. Lastly, the 7-segment display is used to show the musical note information by showing the letter of the corresponding note.

Implementation Description

Initialize and Configure IP Blocks

GPIO configuration is handled by first initializing the device pointers 0 through 3 and setting their default configurations. Then, GPIO direction is set for each bit where 1 denotes a tri-stated input and 0 denotes an output. Since not all bits in the GPIO register are used, GPIO mask defines are used to define which bits are used in the register: 1 indicates the bit is configured, 0 indicates the bit is unused. For example, a GPIO_ANODE_MASK of 0xF indicates that only bits[3:0] are active and being used which will drive the an[3:0] pins on the FPGA device. The timer used is configured to increment on each CPU clock cycle: 100 MHz or 10 nsec period. It is initialized, verified to be operational, and then configured to count up once every clock cycle.

Check Switches and Buttons Services

GPIO0_SW_MASK and GPIO2_BTN_MASK are used to process only data from input pins that are configured. The parameters next to the GPIO0 and GPIO2 device pointers, GPIO0_SW_CH and GPIO2_BTN_CH are their corresponding channel numbers. Changes are detected in the switch and button configurations to trigger an update in the LED and AMP2 states within the polling loop. The read function XGpio_DiscreteRead is used to read the GPIO inputs which takes the GPIO device pointer and the GPIO channel number, GPIO0_SW_CH or GPIO2_BTN_CH.

Calculate Note Service

After checking the switches and buttons, the calc_note service is used to set the frequency and cathode outputs based on the corresponding switch and button data. Accomplished through switch statements, each frequency is mapped to a specific switch and button configuration.

Update AMP2 Service

This service implements the target count timing method which first reads the current count in timer where last count is the timer count at the last toggle. If the current count minus last count is greater than or equal to the target count, then the AMP2 audio signal is toggled and last count is updated with the current count value. The write function XGpio_DiscreteWrite is used to write the GPIO outputs which takes the GPIO device pointer, the GPIO channel number GPIO3_AMP2_CH, and the AMP2 data to be written.

Update LED, Cathode, and Anode Services

The update LED, update cathode, and update anode services all use masks to only update configured pins. For example, GPIO0_LED_MASK is used to update the LED pins which are the configured bits[15:0]. Similar to updating the AMP2, the write function is used with the corresponding GPIO device pointer, GPIO channel number, and LED, cathode, or anode data to be written.

Play Tune on Power-up or Reset

Before the infinite polling loop, a power-up tune was created by polling the current time collected by the timer and updating the notes frequency based on that time. The length of the tune was set to be 2.5 seconds which corresponds to timer count of 250000000, given a 10 nsec period timer. The tune consists of 5 notes, each 50000000 timer counts long, which corresponds to 0.5 seconds per note. Within each timer count interval, the frequency is set, the target count is calculated, and the update AMP2 service is called to produce the sound for each note.

Minimizing Loop Delay

Loop delay is the time it takes to complete one full service loop. Since, the software is architected to run successfully given a maximum delay is not exceeded, the loop cannot be blocked for extended periods of time. If any of the servicing actions described above violated the maximum delay, they would need to be broken down in to smaller steps and executed over multiple loops. Fortunately, this was not the case for the service routines used in this lab, as no routine blocked for significant amounts of time. One precaution taken to minimize loop delay was to not use sleep functions, since these only add to the loop delay. If timing was an issue in this design, a method for tightening timing control would be to change the division of labor in the service timer. Another register could be programmed with the target count value to implement a counter with a programmable terminal count, providing one clock cycle of accuracy rather than the time it takes to complete a full loop.

Conclusion

I preferred working with a high level, general-purpose programming language like C rather than the hardware description language Verilog. I found implementation in C to be more straightforward since the low level details are abstracted away and functions provide additional flexibility. However, I understand that an advantage of Verilog is that operations can be done in parallel in ways that programming languages like C cannot, since C programs are sequential by nature. Challenges faced in implementation included preventing the AMP2 from outputting any noise when no configured switches or buttons are pressed. I learned that a solution is checking for a default target count value of 0 in the infinite polling loop which allows selective updating of the AMP2. I honed my skills in C programming through pointer implementation, dereferencing and referencing variables. As a suggestion, I think that headphones should be provided, since the older headphones required for this lab are much less common nowadays. I particularly enjoyed the power-up tune portion of the lab, as it tested my knowledge of the timer.