Lab 1b Handout

Stopwatch

1 Overview

In Lab 1b we will design and implement a stopwatch using the Seven Segment Display and the Push Buttons on the development board. Most embedded systems by default have to interact with external events. Ideally, these events would be predictable and synchronous, but in many cases, they are not. Therefore, the embedded system must be able to deal with an event at an arbitrary time without affecting the code that is already executing. **Interrupts** are one method of dealing with this problem. Please be aware that this is time-dependent interrupt code development, please start the lab as soon as you can.

2 Hardware

The hardware to be used for this lab is the same as the one that you built in Lab 1A. There is 64 KB of BRAM available to you as well as 128 MB of the DDR. In this lab, you will need to add GPIOs to control the 7-segment displays (more information later – and in the card reference), and GPIO buttons with interrupts. You will also need to understand the behavior of the interrupt controller and timer peripherals. When in doubt, each of these devices has online data sheets describing behavior and signaling. (Access these data sheets via the AMD/Xilinx Document Navigator) or by clicking the block design and selecting "data-sheet".

2.1 Add 7 Seg IP to Project

1. Download the lab files. In Vivado, open Tools > Settings > IP > Repository
2. Click the + button. Then add the folder SevenSeg_1.0 for Nexys or SevenSeg_Urbana_1.0. Click apply
3. Click on add IP (the + button) and search "seven". Add Seven Seg IP to the block design.
4. For Nexys: Make seg_out and an_out external and rename the output pins to seg and an

   For Urbana: Make D0_AN, D1_AN, D0_SEG, and D1_SEG external and make sure the output pins match the constraints file
5. In constraints, uncomment seg (seg[] or D0_SEG[] and D1_SEG[]) and anode (an[] or D0_AN[] and D1_AN[]) lines
6. Increase number of Master Interfaces on microblaze_0_axi_periph to 8. Connect s00_axi on the Seven Seg IP to M07 on microblaze
7. Run connection automation and check "all"
8. In Address Editor, right click on /SevenSeg. Click 'assign all'

9. Generate bitstream and export hardware. Remember to check include bitstream

3 Project Settings

Create a new application project in Vitis. It might be helpful to copy the working code from lab 1a to get started. Although it is not closely related, the 'extra' function provides a timer based interrupt so can act as a starting point. Make sure to generate the linker script (Select the project and go to Xilinx $->$ Generate Linker Script) with the Code, Data, Stack and Heap segments all mapped to the DDR2 (mig_7series_0_memaddr...) and **not** the BRAM. Also make sure that the the heap and stack sizes are 4096 (4 KB). You might need to increase the heap size further if you use malloc() in your code.

4 Writing Software

In this lab, you must present output on the 7-segment LED display. The 7 segment display is multiplexed. This means you can only light one digit position at a time, and must alternate between them quickly to show all the digits of a number. Specifically, you must turn on the anode of exactly one digit position and set the segment cathodes to the correct value for that digit, then move on to the next. This alternation must be repeated quickly to take advantage of persistence of vision, and quickly enough to appear static. A display rate of 20 displays per second might be (barely) legible but will give people headaches. 50-60 times per second or even higher is much more comfortable, some people will still see noticeable flicker to beyond 120Hz. You also need to insure that each segment is activated for the same amount of time (or it will appear brighted or dimmer than others).

An updated LED driver is provided to do the bit packing associated with drawing a digit, (or you can write this yourself). Add the two files to your project, and read the header file for details on the functions. Either way, you have to write the code to alternate between drawing different digits rapidly. Grand loop is a common way to achieve this. Also remember that the display must be functional regardless of what state the counter is in. As a simple observation, the time you spend on each digit should be the same – if you have a bright digit or a flickering digit, look at what determines the time from when it is enabled to when it is disabled (presumably to the next digit). A common software error causes the last digit to be especially bright or dark...

The stop watch is controlled by 4 push buttons which are read in using a GPIO, just like the LEDs are output, however, you need to know when to look for an update. The hardware is configured to assert an interrupt when any change happens to the buttons (under software configuration control). Beware that buttons have contacts that bounce – so that you will likely find multiple assertions of the interrupt for a single depression. This is not a problem is if the button behavior is not modal - i.e. if subsequent assertions do the same thing as the first. In such a case, the loop behavior will be tied to either the first or last bounce of the switch (making a slight timing error). More on de-bouncing later in the lab.

The program will work as follows:

- The seven segments will count up from 0, and all eight segments must initially display 0. Use the upper four digits to display seconds (9999 - 0) and the lower four digits for fractions of second.
- Time measurement counting is controlled by a hardware timer with a fixed frequency. Start out with a frequency of 2 Hz, so that it displays a new value every half a second. Use the upper 4 seven segments of each watch to display the seconds count. Use the lowest 4 seven segments to display fractions of seconds. It is up to you what resolution you will support (i.e. 1mS, 100uS...), but you must have a working version that correctly operates to 0.1s.

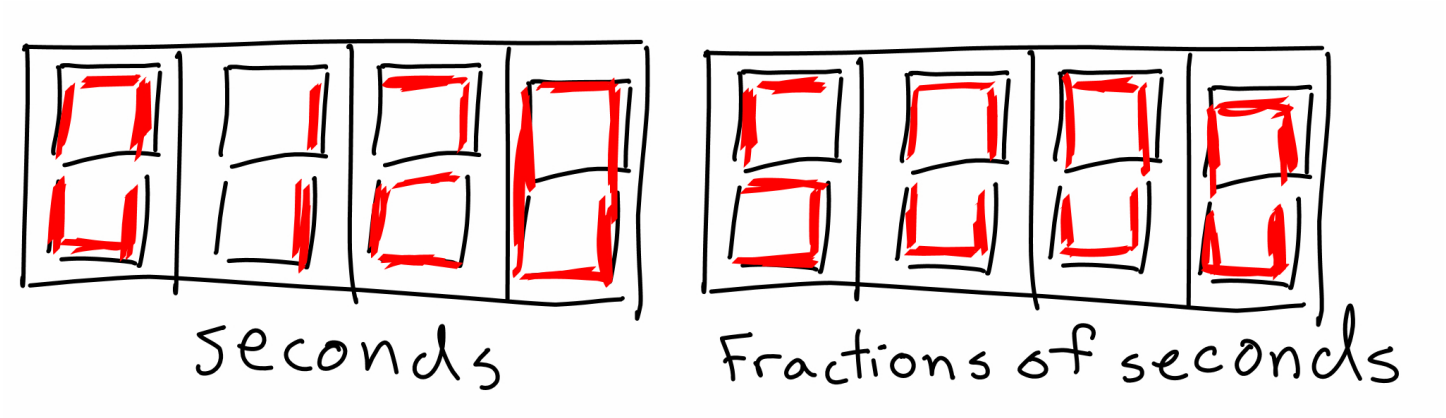

Figure 1: The sketch shows 120.5 seconds

After your stopwatches are working, experiment with increasing the frequency of your timer. How small of time increments would it be possible to display for your stopwatch? Consider the timing requirements for displaying the Seven Segment Display.

You will need to use 4 of your buttons as described below:

- When a **reset** push button is pressed, the stopwatch will reset to all zeros.
- When a **start** button is pushed, the stopwatch will begin timing.
- When a **stop** button is pushed, the stopwatch will freeze the time and display the result.
- When a **count up** button is pushed, the stopwatch will count up.
- When a **count down** button is pushed, the stopwatch will count down. If the stopwatch is at 0, this will do nothing as it cannot count lower.

This lab is simplest if you implement and test functions incrementally. Please do not write a lot of code and then start debugging! One possible sequence is the following:

1. Modify the lab 1a code to print a character to the terminal every 1/2 second using a timer interrupt. (This means you have understood how to make an interrupt work off the timer).
2. Edit to draw a fixed digit to the 7-segment led bank, updating to the next digit every 1/2 second. Then, speed up the interrupt until the digits are evenly lit and flickering disappears. Be sure the digits are all the same brightness. Finally, have the display depict the value of an int variable. This should be done by making a simple grand loop and editing it. (You now have a working 7-segment display).
3. Implement the the running counter function on the display using additional timer interrupt to increment grant loop variables. What happens if the counter runs faster than the display loop update?
4. Implement a push-button 'clear' function using a button interrupt which clears the running counter above. Note that there should not be visible 'glitches' in the display from the button depression or the running counter.
5. At this point you have all the basics needed to finish the stopwatch using the other 4 buttons.

To learn more about how to implement the interrupts read the code included in extra_method.c and extra_method.h. The auto-generated test code in the "peripheral test project" is also helpful. The documentation included on Canvas for **Xilinx Interrupt Controller Documentation** is a good resource. This project uses interrupts from two different sources, the AXI Timer module and the AXI GPIO module. Both types of interrupts use slightly different programming implementations.

Tip: Create a 'test peripheral' project in Vitis to see how Xilinx implements interrupts according to your hardware configuration.

NOTE: For debugging an interrupt handler, use a variable to turn an LED on and off. Don't use print() – it may well cause errors and since it is buffered, you can't tell if it passed or not during execution.

4.1 Button Bouncing and Glitchs

Remove the count down button and make the count up button into a toggle that changes the counting direction with each press. Play around with it by trying lighter and harder presses. Notice any odd behavior? Slight changes in how the button is pressed and released can make the switch bounce. Since the button is linked to a (modal) toggle, this causes unwanted state changes. Does the glitch happen on depression or release (or both)?

Buttons are physical switches with internal springs at relatively high tension. When they are switched, nearly all buttons will make a contact, but then release it (possibly several times). The time between bounces depends on the tension and the button contact mass, but is usually between 30-300uS. The effect is that the interrupt you capture, may well be repeated multiple times. If you saw glitches in the modified program, please explain why you didn't see them in the original configuration.

4.2 4 Buttons for 5 Functions

When you only have 4 buttons available what should you do? We recommend implementing the forward / backward count modes as a single push button. For example, if the stopwatch is in forward mode, when you push the button it should switch to backward mode. How would you debounce the button?

Testing suggestion: Turn one of your LEDs on / off to indicate the direction the count is.

Debouncing hint: The timer can be useful to debounce. A real human isn't going to press the button as fast as the button bounces.

5 Reporting

Within two pages, explain how design decisions for using a grand loop and interrupts changes the functionality of your stopwatch.

1. How fast of a stop watch are you able to build using your hardware and software configuration? What was the limiting factor in the accuracy of your stopwatch?

2. How did you control the display update timing (i.e. loop period) and did this effect accurate timing measurement? If so, how? What display update period did you choose?

3. The push buttons in this stop-watch are implemented using interrupts. Would it be possible to instead check the values of your push buttons while executing the grand loop? Would this polling approach change the timing of your stop watch?

4. What will happen if two push buttons are pressed at once?

5. Describe the UI of the stopwatch. List the input and output devices.

6. Describe anything unique that you did to make your stopwatch work better.

6 Report Photos

In your report, take two photos of the working timer. Take one photo after some time has elapsed and the timer is stopped. Take the other photo while the timer is counting. In both cases, is the illumination of the digits constant or does it seem to vary with the displayed code or favor one or the other digits? Why might this be the case? Remember that the perceived illumination is dependent on the timing and possibly on the possibility of multiple digits running at the same time. Both start and stop times matter for even illumination.