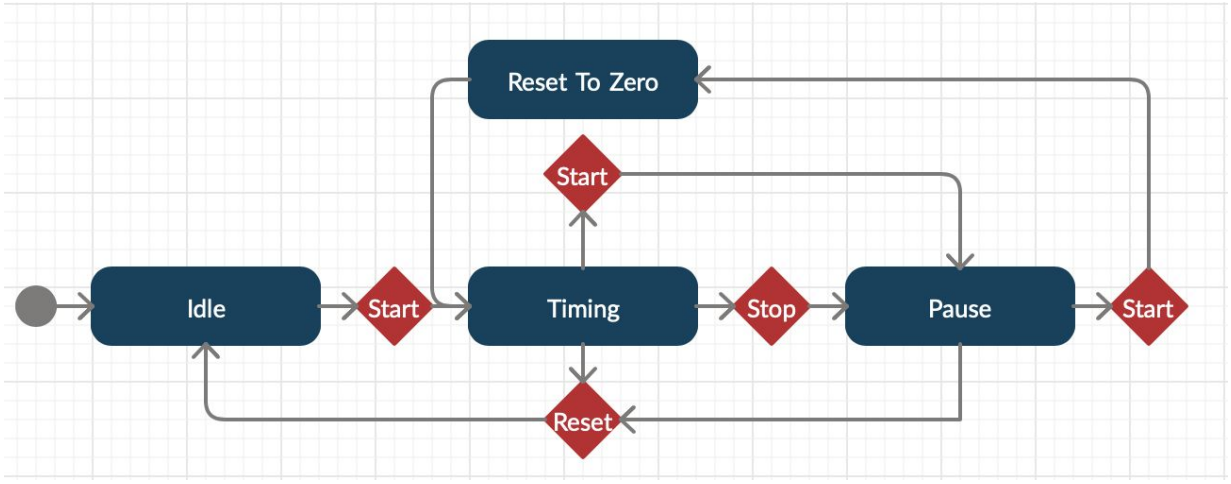
ECE 153A Lab 1B: Stopwatch

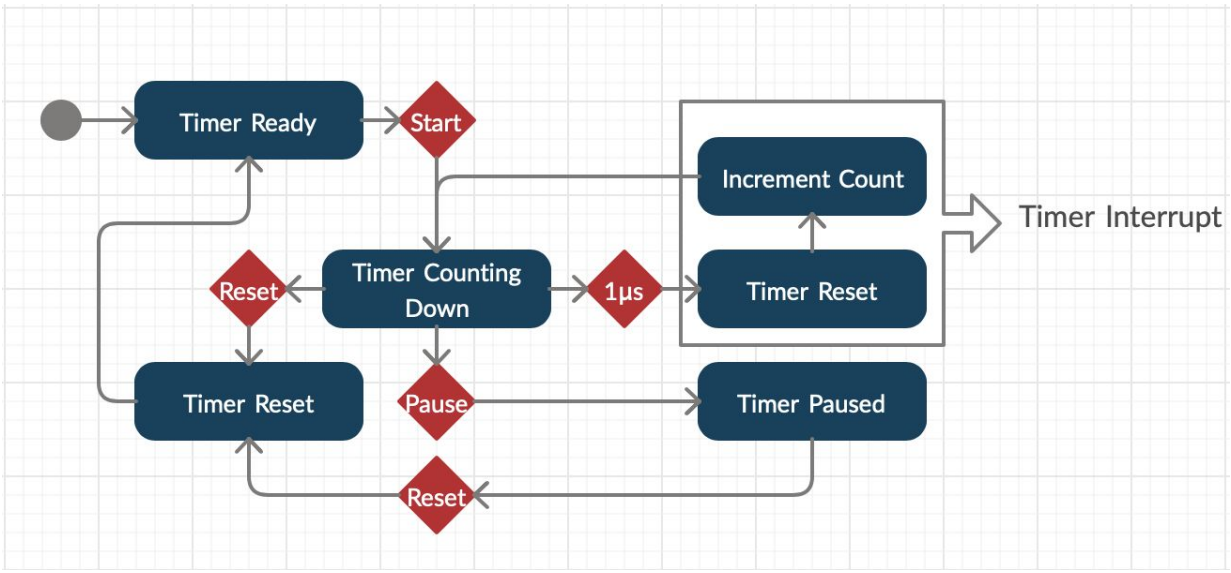
Luke Lewis | Eduardo Olmos

Purpose

In this lab we will be using interrupts and hardware timers to interface the Seven Segment Displays and the Push Buttons peripherals on the Nexys A7 FPGA Board and implement a stopwatch. The stopwatch must be able to provide accurate time measurements of resolution greater than 0.1 seconds. This requires us to update the seven segments displays at high speed to match time transitions, for our case we will be attaining a resolution of microseconds. Additionally, we need to implement three controls per the push buttons: start, stop and reset.

Flow





Results

The stopwatch consists of three inputs, which are three separate buttons each doing a separate task: A reset button (resets the stopwatch to zero), a start button (starts the stopwatch timer when at zero or resets when paused), and a stop button (freezes the stopwatch time and displays the result).
The outputs consist of eight 7-segment LED displays, which are purposed together to form a single functional display for the stopwatch. The three input buttons will directly affect the output display.

For our implementation of the stopwatch we were able to capture a resolution of 100 microseconds. This was accomplished with the clock frequency of 100MHz and a timer interrupt. The timer was set to a value of 100000000 / 10000 = 10000 cycles and set to decrement every cycle. So every 10000 cycles which occurs at 100 microseconds, an interrupt is triggered and a counter is incremented. The limiting factor in the timing accuracy of our stopwatch is that it is based on a timer counting down cycles. Once we hit the timer interrupt, there is work to do. This work is done very fast but is still an amount of time that in between each interval of 100 microseconds interrupts. This means our timer will drift slower in a cumulative way. This could be improved with auto reloading option for the timer so the timer clicks while inside our interrupt handler.

We noticed that trying to increase the resolution to 10 microseconds by setting the timer to 1000 cycles would cause the timer to run slower than real time. We suspect this is because the interrupt is firing too fast it is missing interrupts. To control the display update timing we simply ran it as fast we could.

It would have been possible to check the values of the push buttons while executing the grand loop, however the reads would only happen at the beginning of the loop. This means that it makes timing less accurate as you could hit the push button and not have it read until the next iteration of the loop. Furthermore, the duration of the button press could make for bouncing issues as it could be longer than the duration of the loop. This would cause a single button press to be registered as multiple presses.

The behavior when two push buttons must factor that it’s highly likely that there will be a tiny amount of time between the two buttons pressing, but that is enough for the processor to recognize the very first button press, and then disable future interrupts (second button press) until the first interrupt is complete. This is debouncing, where we prevent short bursts of consecutive reads when a button “bounces” when we press it. If we can somehow make the two buttons activate at the same time, then reading from the GPIO would give us a value different from the values of a single press of either button. The GPIO input sets a bit high to be representative of a particular button press. If both are activated at the same time, then both would be set so that the particular bits representative of both buttons go high. We defined our

behavior for only one bit to go high and thus no action would be done as our logic didn't target that particular input. We could improve upon our design and specify precedence behavior but we did not do this in our design.
To improve upon our design, we implemented debouncing through the use of disabling interrupts. When the GPIO interrupt gets triggered, we immediately disabled the interrupt. We read from the GPIO input and then call sleep for 300 microseconds. Afterwards we clear the interrupt and enable it. This allows us to have 300 microseconds of time in between presses and stabilizes our input. We also tried to enable and reset our timers in the correct places. This means making sure our timer is reset before we start it for the first time once we press start. As to not have it count immediately after it is initialized. In the timer handler, we reset the timer only after we have incremented our counter. When reset is pressed, we first disable the timer first and then we zero out our counters.

Conclusion

In conclusion, through this lab, we were able to get a feel of using basic I/O on our board by utilizing its hardware buttons and LED outputs in an implementation of a stopwatch. From this, we found that interrupts are necessary in order to get us as close as possible to a real time response to button pressing, and also a useful tool to implement a reliable timer with great resolution by interrupting at specific cycle increments reached by the board's built in clock frequency.

The issue of 'bouncing' was introduced in this lab, and a workaround ('debouncing') was necessary in order to keep the stopwatch from having issues. What we learned from this is that even a seemingly straightforward embedded system with 3 discrete button inputs can easily be prone to issues which must be addressed in order to ensure that the system is reliable. This is of utmost importance when designing embedded systems where a slight error can result in dire consequences.