Lab 2A Handout

The Rotary Encoder

Lab Overview

In this lab, we will learn how to implement a finite state machine (FSM) based interface. The state machine uses information provided by the peripheral (the rotary encoder) to trigger state changes and generate events. The goals of the lab are to provide a cleaner debounced interface for the quadrature encoder device, which has a well deserved nasty reputation of high debouncing complexity.

1 Introduction

You will modify the Vivado project used in Lab 1B by adding two GPIO devices and recompiling the hardware. (Be sure to close the SDK before recompiling the Vivado project as you do not wish to create more than one Board Support Package). The rotary encoder is connected to the FPGA board via the Pmod connector JD which is connected inside the FPGA via a GPIO device on the AXI peripheral bus. The rotary encoder is used to explore software debouncing of peripherals (In this case, FSM based debouncing). The debounced rotary encoder then controls an LED display, so the proper behavior of the encoder can be demonstrated. You will also connect the push button of the encoder to use as a display toggle. In this case, you will need to implement timer based debouncing of the push function. Finally, the 'idle' process will be a simple flashing green light that indicates the software is running properly. It is often a very good idea to program tell-tale indicators as they are a nice debugging aid.

2 Procedure

1. Create an output GPIO for the RGB LEDs (6 bits). Creating GPIO AXI peripherals is described in the Hardware Handout in 'General Resources' on Gauchospace. The GPIO peripheral must be connected to the proper callouts in the xdc file. The default callouts for these leds is LED16_B or LED17_G etc. Please edit the xdc file callouts to rename these pins as rgbleds[0-5] for all pins. This allows simple connection to a single GPIO output device of 6 bits.

2. Create an input GPIO to service the JD Pmod. You should make the GPIO 3 bits wide (unless you wish to connect the slide switch as a fourth pin– not used in this lab). Enable interrupts on the GPIO module and connect the 'Interrupt' pin to the Concat module. You will need to increase the number of ports on the Concat module. Name the GPIO module "Encoder" so that you know how to refer to it in the code. The behavior of the encoder as wired defaults to high for the outputs A and B, while the push button is default low and pulls high when active. For the sequencing of A and B, see the handout "Notes on Mechanical Rotary Encoders and Debouncing" in this week's section of Gauchospace. You can use your Homework 3 solution for implementing debouncing. Note also that you will have to renumber the pins to start form zero in the constraint file since the block diagram insists on numbering from zero. Unfortunately this

breaks the relation between number printed on the board and number in the block diagram. The pins in your block diagram and .xdc file should be named JD[0:2]. These correspond to pins 1-3 of the Pmod connector as shown in Fig. 1.

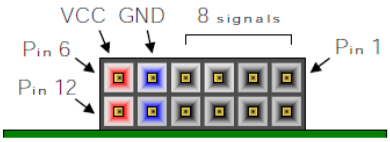

Figure 20. PMOD Connectors- Front view as loaded on PCB

Figure 1: The pin numbering of each PMOD connector.

3. The hardware debugger in Vivado can be used to probe signals in your hardware and monitor their values, hence simplifying the debugging process. Click 'Run Synthesis' in the Flow Navigator. Open the Synthesized Design, and click 'Set Up Debug'. Add monitors to the following signals:

   - JD_IBUF (the JD pins after the input buffer in the IO pad of the FPGA).
   - system_i/microblaze_0_local_memory/ILMB_abus (the processor address bus for instruction fetches).
   - system_i/axi_gpio_Encoder_ip2intc_irpt (interrupt singal from GPIO to interrupt controller).
   - system_i/microblaze_0_interrupt_INTERRUPT (interrupt signal from interrupt controller to CPU).

   While on the choose nets screen, you can use the "Netlist" tree view in the main Vivado behind the window to choose these nets. Set the sample depth to at least 4096 (upto a maximum of 65536). Save the updated constraints file, again run Synthesis, Implementation and Generate Bitstream.

   Note: One known issue with the debugger is that if you remove a net that is set to be debugged the compile will fail. If this occurs, remove and re-create the debugger.

4. Use the "Open Hardware Manager" in the Flow Navigator of Vivado. You can create a trigger using one of the available debug signals in the Trigger Setup Window. Choose a signal to trigger on and the condition to trigger on. Press "Run Trigger" and once it switches to "Waiting for Trigger" go to the xSDK, run your code, and switch back to Vivado, to see a waveform from the debug probe.

5. The remainder of the project is purely software development (assuming that the checks made above work correctly). Export your hardware and launch SDK. Update the board support package if you are making changes to you existing Lab1B project for this lab. Also run the linker script and make sure that all segments of the memory are mapped to the DDR (mig_7series_0_memaddr...) and **not** the BRAM. Also make sure that the the heap and stack sizes are 4096 (4 KB).

6. Create a simple software loop to make a blinking green light on one of the rgbleds. This code should be simple wait or loop delays as it will be interrupted by the other parts of the code later. This loop is always running by default and indicates the status of the board. It also checks to make sure you know how to connect the GPIOs.

7. Create and code a software project that controls the 16 LEDs on the Nexys board according to GPIO interrupts from the rotary encoder. The overall behavior required is as follows:

   - Turning the encoder clockwise will move the illuminated LED to the right. At the right-most position, wrap to the furthest left position.

- Turning the encoder counter-clockwise will move the illuminated LED to the left. At the left-most position, wrap to the right-most position.
- Pushing the encoder down will do the following:
  - If the LED is illuminated, turn off the LED.
  - If the LED is off, re-illuminate the LED in the position it was before it was turned off.
- If the encoder is turned while LEDs are off, nothing should happen.

Break this behavior into two groups of procedures: The first comprises simple routines to update the LEDs as function callouts (e.g. led_left(), or led_off()). These routines will be called from the idle procedure at the behest of shared static flags. The second will be the debouncing (for rotary encoder twist) and timing (for debouncing rotary encoder push button) FSMs in the interrupt handler. Both FSMs set static shared flags that are read by the idle loop to update the LEDs.

The rotary encoder idles at 11, and when it rotated it goes through the sequence 11, 01, 00, 10 and back to 11, or the reverse sequence for motion in the opposite direction. The debouncing FSM must recognize this sequence and set a flag when the full sequence has been reached. Unfortunately, the mechanical contacts in the encoder will not smoothly transition from open to closed or vice-versa. Effectively, this means that a measured sequence might look like {11, 01, 11, 01, 11, 01, 00, 01, 00, 10, 11, 10, 11 }. Build your FSM so that these extra transitions move back and forth between the states. After the sequence has been found, set the flag on return to the idle state (11 input) of the FSM. In your write-up, draw a bubble chart clearly showing the states and inputs and actions designed for your solution.

The GPIO controller for the encoder inputs are wired for interrupts, and given an appropriate interrupt enable, it will call the handler each time the inputs change. Note that if you need to read multiple flags in the idle loop to take action, be sure to make the read atomic (or the interrupt can happen in the middle, causing a bug).

Debouncing the push button requires a timer or timing loop (Why?). In this lab, it is simplest to have the interrupt handler set the flag on each interrupt as well as copy the current time from a pre-running timer. Then, in the idle code, you poll the flag and the timer and wait till enough time has passed that bounces have ceased. This method of timer use allows for multiple processes to share the timer. You could make the timer interrupt driven as well, however, this opens a number of potential problems with shared data and sequencing as well as potentially using more stack space. (It is a potential problem on stack depth and on volatile registers depending on the crt0.o dispatch code.)

A good grade in this project requires that the system works robustly even when the knob is turned rapidly and similar fast/simultaneous inputs. Pretend your 3-year old sibling is playing with it...

Note: If you see that the interrupts initialization fails often on multiple runs of your code, check "Program FPGA" in Debug Configurations. Your FPGA will be re-programmed for every new debug session and this will ensure all stale cache values are reset.

3 Lab 2 Files

Please make an effort to eliminate extra code in your source files that you have accumulated during Lab1A and Lab1B. Extract only what is necessary from your old code (initializing timers, GPIO, interrupts, etc). It would be a good idea to start with a fresh software project.

4 Reporting

Write a two page report, detailing the steps you took to create interrupt handlers for the rotary encoder.

1. Explain how you decode which direction the rotary encoder is twisting in. Also explain how debouncing for the twists of the encoder was done. Show bubble charts (states, inputs and transitions) for the finite state machines in your design.

2. Explain how debouncing for the click of the encoder was done.

3. Use the hardware debugger to record the a waveform of the encoder GPIO change interrupting the processor.

   - Set a rising edge trigger on the ...Encoder_ip2intc_irpt interrupt signal in the Trigger Setup Window
   - Hit capture (the play button) and turn the knob with your code running so an interrupt actually happens.
   - In the captured waveform, you should be able to see the JD values changing, then the two interrupt signals changing. Then there is an abrupt change in the address lines to 0x00000010 corresponding to the processor jumping to the overall exception handler, which then filters through various drivers to your code.
   - In SDK, opening Project/Debug/<project_name>.elf will give an assembly listing of your program. You can search either function names to find their addresses, or you can search addresses you see in the HW debugger to see where in the program they are (omit leading zeroes when searching addresses).
   - You can find when the code goes from drivers into the interrupt handler you wrote by finding the first address of that function in the assembly listing, and then using the "Find Value" function in the hardware debugger.
   - Measure the delay in cycles from input change to GPIO interrupt, from GPIO interrupt to processor interrupt, from processor interrupt to address changing to 0x00000010, and from that to the entering your interrupt code.

   Include two screenshots. **(A)** The interrupt from the push button and the GPIO signal. **(B)** The interrupt from the twist button and the GPIO signals (there are 2) for twist.

4. What is the Microblaze MSR Register used for? What is the purpose of Bit 30 in the MSR? (Reference: MicroBlaze Processor Reference Guide)

5 Demo Guidelines

Each group can work together on the lab, but must write report and do demo individually. You will need to be able to demonstrate the following:

1. Turn the knob of the rotary encoder in clockwise direction slowly (approximately 3 seconds for one rotation). Also capture the wrapping of the on LED to the left-most position.

2. Turn the knob of the rotary encoder in anti-clockwise direction slowly (approximately 3 seconds for one rotation). Also capture the wrapping of the on LED to the right-most position.

3. Repeat 1 and 2 by turning the knob rapidly in each direction.

4. Finally, press the push button on the encoder. Now twist the encoder in each direction several times, then press the push button again and repeat twisting in each direction.

Appendix: Microblaze interrupt signalling

The Microblaze processor uses a level based interrupt model, that is whenever a peripheral has an event occur that should cause an interrupt, it raises its interrupt line high and holds it there. Similarly, the interrupt controller sends its one interrupt to the Microblaze processor in level sensitive fashion. Whenever the interrupt line is high and Microblaze has interrupts enabled, an interrupt is started. The interrupt handler should make sure to acknowledge (or clear) the interrupt back in the peripheral, so that the level goes back low and processing can resume. Otherwise the processor will be re-interrupted the instant it returns from that interrupt, thus re-enabling interrupts in the processor. This is also why interrupts are disabled by the processor when it starts processing it, otherwise it would just get stuck in a loop of starting to process the same interrupt again and immediately getting re-interrupted. It is also useful to know this when examining hardware debugger traces.