

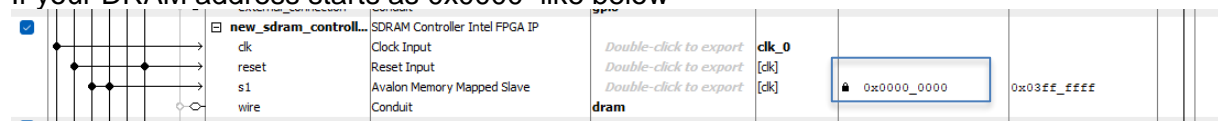
LAB 4 : NIOS interrupt and comparison with polling

1. Event Handling with Interrupts

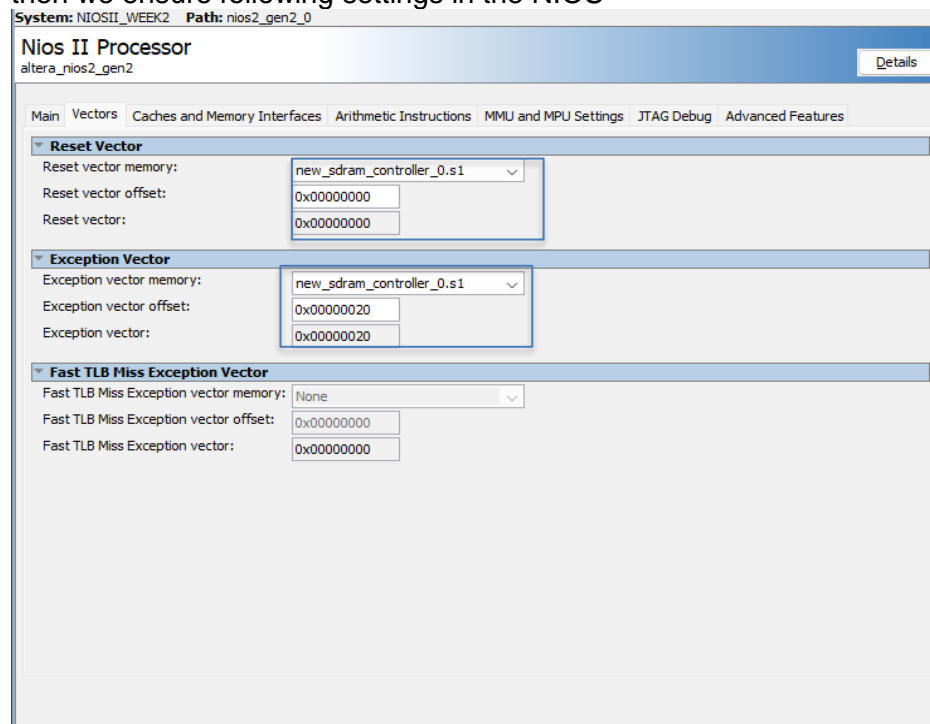
Now we will modify the program to use interrupt-based event handling instead of polling. Compared to the method of implementing interrupts in the workshops, we will actually use a lower-level implementation to enable interrupts. The higher-level implementation will be covered during the workshops and the RTOS-based labs, where you will skip some of the lower level details in enabling interrupts.

Step 1:

If your DRAM address starts as 0x0000 like below



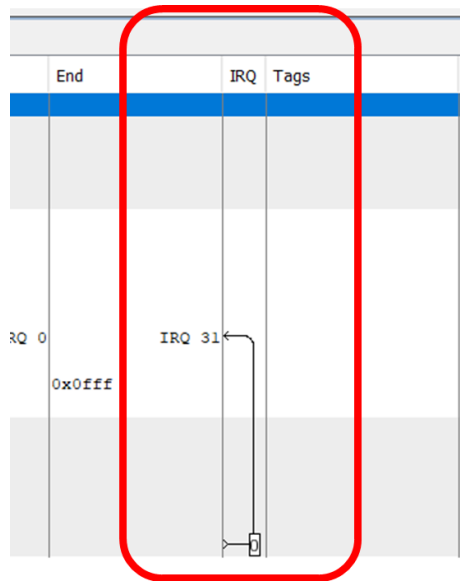
then we ensure following settings in the NIOS



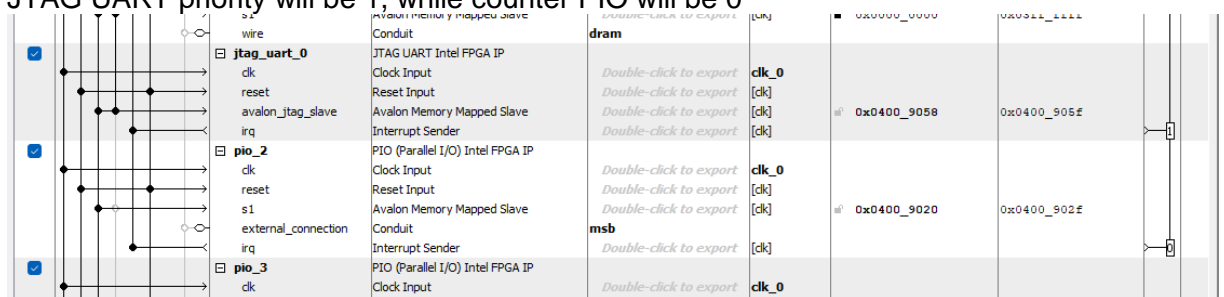
Step 2 :

To enable hardware interrupts on the PIO, you will need to add the IRQ to the PIO via Platform Designer by:

- Double-clicking on the PIO that you want to enable the interrupt for
- Tick “Synchronously Capture” and “Generate IRQ”
- In the IRQ Type, select “EDGE”. If you cannot see the option, click the drop-down menu next to IRQ Type
- In Platform Designer, double-click the bubble in the IRQ column corresponding to the PIO. See figure below for more information



- After this, you will need to update your Quartus and Eclipse project by following the [ECE3073 Help Manual.docx](#) in section “4.2 - Reflecting Quartus changes in Eclipse”
- JTAG UART priority will be 1, while counter PIO will be 0



1.1 Setting up Eclipse for low-level interrupt implementation

Due to the various compiler directives that you can provide to the Nios II processor, you need to change the steps to setting up the Eclipse program for this part. We highly recommend to name this lab2_interrupt so it is easy to keep track of your projects. To setup the Eclipse program:

1. Download the skeleton codes for the interrupt section [here](#). There are four files:
 - Makefile - You do not need to edit this unless you run into the build error (step 4)
 - main.c - This contains the main C file where you will be completing your interrupt initialisation code
 - exception_handler.c - This is where the exception code will run, and jump into your interrupt_handler function if the interrupts have been initialised correctly
 - nios2_ctrl_reg_macros.h - This is where the NIOS II macros are to write to the control registers within the Nios processor. You will use the functions from this header file in your main.c and exception_handler.c
2. Create an empty project with “File > New > Other > C/C++ > Makefile Project with Existing code” shown in the left of figure1 and set up the project as in the right of figure 1:
 - In “Existing Code Location”, browse to the folder, named “lab2_interrupt”, that contains the downloaded files for lab2_interrupt

- Use the following settings (see figure 1):
 - i. Name the project the same name as the folder, in this case we used “lab2_interrupt”
 - ii. Languages: C only.
 - iii. Toolchain for Indexer Setting: “MinGW Nios II GCC4”¹
- 3. Click **Finish**

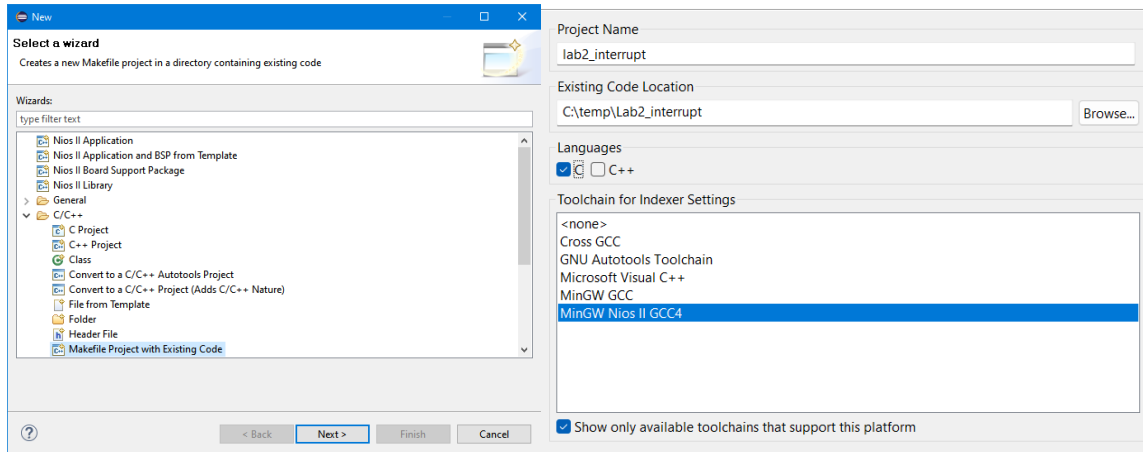



Figure 1: Project Setup for low-level interrupts

- 4. Click on the “Build All Icon, , or press “Ctrl + B”.

If you get a “linker script” or “make” error, please check your installation directory of “C:\intelFPGA_lite”. If your directory contains “lite”, then you will need to update the “makefile” in the “Project Explorer” tab. Double click the “makefile” in Eclipse, and replace “intelFPGA” with “intelFPGA_lite”. This should be in lines 20, 21, 23 in the makefile.

¹ otherwise the IDE will complain that I cannot find “make” binary

You are provided with skeleton code for a normal code main function and interrupt code. The following figure shows flowcharts for this code. Your C code should implement the flowchart elements in the dashed boxes. You need to refer to your lecture notes and the Nios embedded peripherals datasheet on Moodle in order to learn how to set up and service an interrupt.

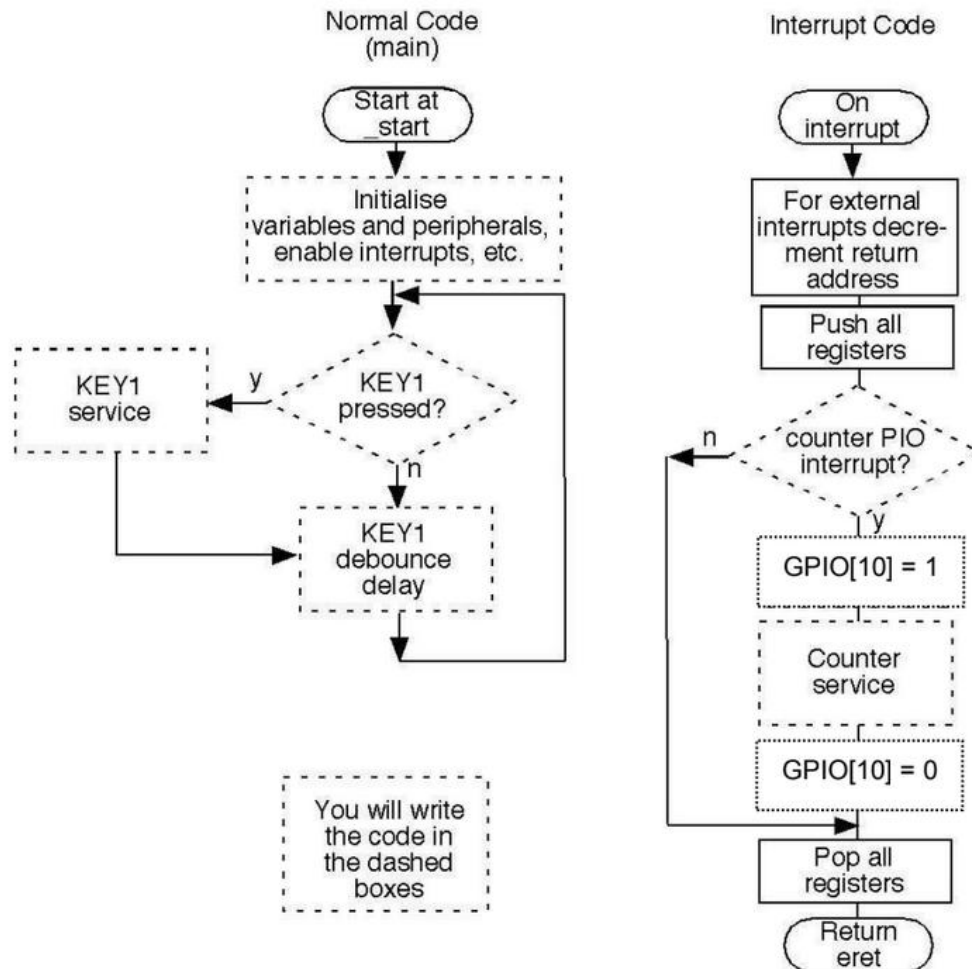


Figure 2: Flow chart for interrupt code triggering if an event (MSB) has occurred

A good way to troubleshoot your interrupt system is to place a breakpoint where your interrupt handler is. This is to test if you interrupt is:

- 1) Initialised correctly
- 2) Triggered by an event
- 3) Goes into the correct interrupt handler (ie. if you had to service various interrupts)

If your code does not ever enter the interrupt handler, take a step back and evaluate if you have satisfied the four conditions of interrupts:

- 3 condition of interrupts stem from the Nios II control registers
- 1 condition is based on the interrupt mask of your PIO

Task 1 : Demonstrate this part YES /NO

You should see LED's counting the MSB rising edge this time by interrupts.

Task 2 : Theoretical Maximum and Minimum Latency

Once your code is compiled, take a look at the disassembly of the program and calculate the minimum and maximum theoretical latency that you expect for the counter service. For stw instructions, you can assume it takes 9 pulses per instruction to execute². Use the flowchart provided above to help you to determine the minimum and maximum latency paths.

Description	Latency (ns)
Expected latency	$171 * 20 = 3420\text{ns}$

Measuring Latency with Latency Counter Module

While running your program on your board, observe the latency count that is displayed on the hex displays. Take a note of several counts, and fill in the blanks below:

Description	KEY[1] not pressed -
Observed latency (ns)	-

Measuring Latency with DSO

Now we will see an alternative method to measuring latency - using the DSO.

Connect the ground of the oscilloscope to the ground of your board, connect channel 1 of the DSO to GPIO[2], and channel 2 of the DSO to GPIO[10]. Note that at low digital oscilloscope timebase speeds very narrow pulses will disappear completely. Therefore, if you cannot see a short pulse on the oscilloscope you should try increasing the timebase speed (zoom in).

Description	KEY[1] not pressed -
Observed latency (ns)	27.2 us

² This number is actually retrieved from lab 1 when measuring the number of clock cycles per instruction!

Task 3 . Based on Lab 3 and Lab 4 answer the following

(a) What is a major source of latency in an interrupt-based system?

the time it takes to process the interrupt signal, save the current context, switch to the interrupt service routine, and then restore the original context.

(b) Why are we not concerned with minimum or maximum latencies for the interrupt system?

Because interrupt system triggers an exception that will not miss any clock input, hence there is no min or max latency.

(c) Fill the table below, check no. of logic elements

Implementation	Number of logic elements
Polling	
Interrupts	

(d) When using polling, what happened to the latency when KEY[1] was pressed? Why?

(e) When using interrupts, what happened to the latency when KEY[1] was pressed? Why?

(f) How well do the calculated and experimental latency values compare? What might account for any discrepancies between the two?

(d) What is a potential issue you see arising with polling-based event handling in more complex programs?

(e) What is an advantage of using polling to handle events in time-sensitive contexts?

(f) What do you notice about the number of logic elements between the polling and interrupt implementations? Explain why this is the case.