COE4DS4 Lab #3
Interfacing Custom Hardware to the NIOS II Processor

Objective

To understand how to interface custom hardware components to the NIOS II embedded processor using the system-on-a-programmable-chip (SOPC) tool flow; you will also explore performance issues in an embedded system.

Preparation

- Read this document and get familiarized with the source code and the in-lab experiments

Experiment 1

The aim of this experiment is to get you familiarized with the two main ways used to integrate custom hardware components with the NIOS II embedded processor in the Qsys environment. Together with the device driver created in the NIOS II SDK, the hardware components can communicate with the NIOS II processor.

Part (a) Figure 1 illustrates the architecture of an embedded system in which the hardware component is connected through general-purpose input/outputs (GPIOs) for *experiment1a*.

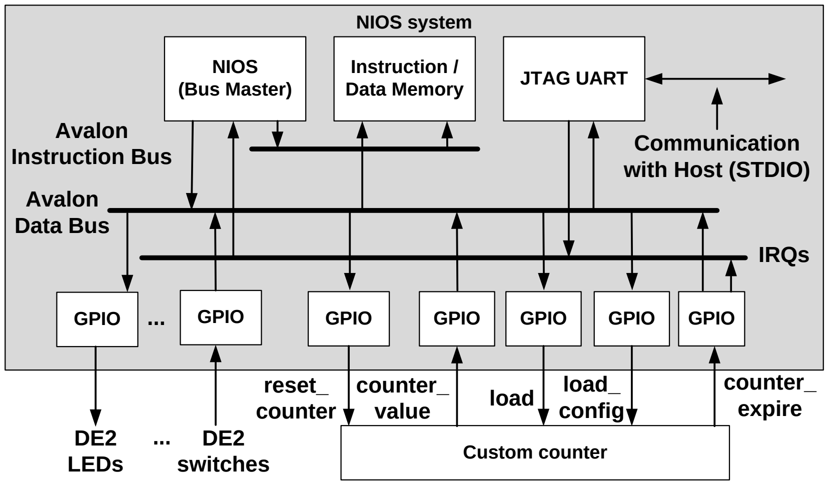

Figure 1: Integration of a custom hardware component and NIOS processor using GPIOs.

In this experiment, a simple counter is used as the custom hardware component. It is setup in such way that five GPIO cores are instantiated to create five communication channels between the counter and the processor. The five channels that are created are: (i) *reset_counter*, which is used to reinitialize the counter to a pre-defined value, as defined below; (ii) *counter_value*, which allows NIOS to read the current value; (iii) *load* indicates that the counter should load the *load_config* value provided on channel (iv); *load_config* is a 2-bit signal defined by SWITCH_I[16] and SWITCH_I[15] as follows: when 00 the counter expires after 1 second; when 01 it expires after 750 ms; when 10 it expires after 500 ms and when 11 it expires after 250 ms; (v) *counter_expire*, generates an interrupt to notify NIOS each time the counter has expired; note, the counter will not restart itself after it has expired until the *reset_counter* signal is asserted through an IOWR to the appropriate address (as defined in Qsys). It should be noted that other GPIOs are also used in the NIOS system in order to provide the interface the standard DE2 peripherals, such as switches, green/red LEDs or pushbuttons.

When using the GPIO cores, each of these five communication channels can be accessed by the NIOS processor using IORD and IOWR to the specified base address under the Qsys environment. You have to perform the following tasks in the lab:

- Understand how the custom counter is connected to NIOS in *experiment1a.v*, and get familiarized with the software to understand how the custom counter communicates with NIOS;
- Note: SWITCH_I[17] is used as an active low reset and the push buttons are used to send the commands (reset, load, ...) to the custom component (the spec can be inferred from the C code)
- In the reference code that is provided, the custom counter is instantiated in the top-level module, together with NIOS system. However, the re-initialization of the timer value is implemented only when *load_config* equals to 00. Extend the implementation in such way that all the four possible values for *load_config* are supported (as described on the previous page)

Part (b) The second way to connect a custom hardware component to the embedded processor is to interface the component directly to the Avalon bus. Figure 2 illustrates the architecture of such system given in *experiment1b*.

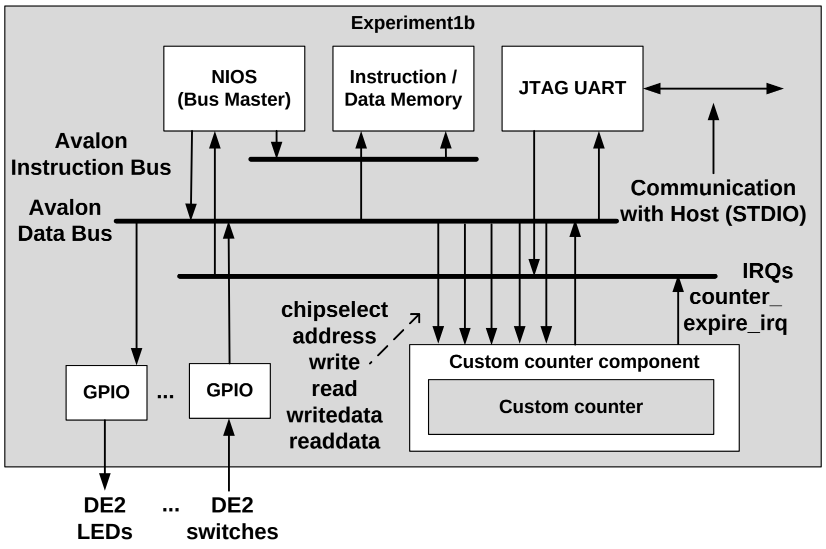

Figure 2: Integration of a custom hardware component directly onto the Avalon bus.

In this experiment, the custom counter is wrapped inside a component module, which describes the connection protocol with the Avalon bus. You should find the following signals in the wrapper:

- *chipselect*: this signal will be activated when the data on the bus is addressed to the component
- *address*: this is the address of the configuration register which NIOS is accessing (this address equals the offset through which an individual register is identified within the component)
- *write*: this signal will be activated when a write request is sent to the component from NIOS
- *read*: this signal will be activated when a read request is sent to the component from NIOS
- *writedata*: this is the data the component receives from the bus during a write request
- *readdata*: this is the data the component sends out during a read request

In addition, the signal *counter_expire_irq* is connected as an interrupt to the NIOS processor in order to notify when the counter expires. It is activated whenever a positive edge of the *counter_expire* signal from the custom counter is detected.

You have to perform the following tasks in the lab:

- Follow the instructions from the "readme_steps.txt" provided in the *experiment1b* folder in order to build the NIOS system shown in Figure 2
- Understand how the custom counter component communicates with NIOS, and understand the conceptual and implementation differences to the design from *experiment1a*

Experiment 2

In this experiment, you will integrate the *PS2_controller_component* that wraps around the provided *PS2_controller* in order to create a system in which the NIOS processor takes inputs from a PS/2 keyboard.

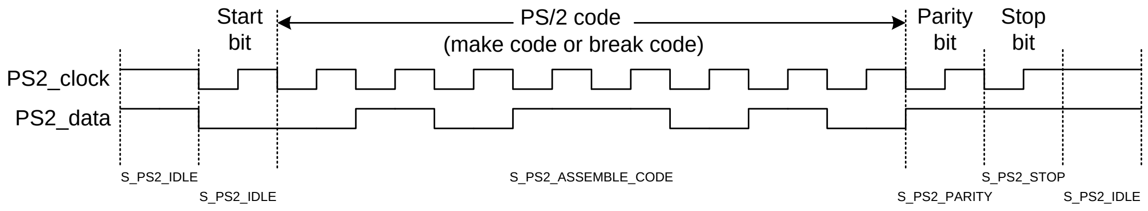

Figure 3: Timing of the PS/2 interface.

Figure 3 shows the timing of the PS/2 interface. This interface has two bidirectional signals: *PS2_clock* and *PS2_data*. The given implementation in *PS2_controller* supports only the receiver part of the PS/2 interface (i.e., we only receive data from the keyboard). If a key is pressed or released 1 byte of info is sent approx every ms from the PS/2 keyboard. There is a reference clock (*PS2_clock*) which is "1" when no data is sent. When the PS/2 device initiates the communication a *PS2_clock* pulse must be accompanied by a start bit (active low). This is detected in the S_PS2_IDLE state from *PS2_controller.* In the next state (S_PS2_ASSEMBLE_CODE), on each edge of the *PS2_clock*, the value of the PS2_data is fed into a right-shift register (PS2_shift_reg). After 8 clock cycles the parity bit is checked in the S_PS2_PARITY state (in our implementation no action is taken on this parity bit, nonetheless the state must exist in order to comply with PS/2 protocol initiated by the keyboard). The final state S_PS2_STOP checks if the stop bit (active high) is passed with the last *PS2_clock*. In the same state we set a flag *PS2_code_ready* that together with the *PS2_code* (loaded from the PS2_shift_reg) and the *PS2_make_code* flag (detailed in the next paragraph) will be passed for further processing outside of the controller. It is important to note that the finite state machine (FSM) from *PS2_controller* is clocked at 50MHz and it is enabled when an edge is detected on the *PS2_clock*.

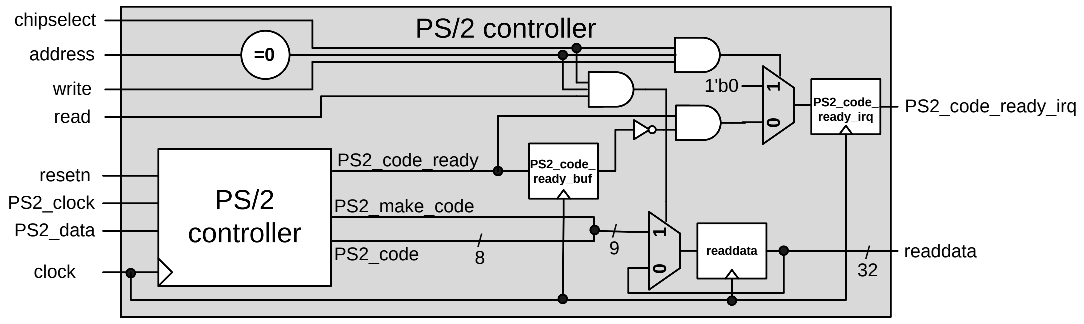

Figure 4: The PS2 component and its interface to the NIOS processor.

Before interfacing the *PS2_controller* to NIOS, a wrapper must be implemented, as shown in Figure 4. Note, when a key is pressed a "make code" is generated and when a key is released a "break code" is generated by the PS/2 keyboard. For the keys that we are concerned with in this lab, the make code is 1 byte and the break code is 2 bytes (the first byte is 8'hF0 and the second one is the same as the make code). The keys supported in our simplified implementation are available in the "PS2_keyboard_codes.pdf" file provided in the *experiment2* directory. The *PS2_make_code* flag that comes out of the PS/2 controller module can be used to differentiate between the make codes and the break codes. If an edge is detected on the *PS2_code_ready* signal, the interrupt signal *PS2_code_ready_irq* will be asserted. The NIOS processor can read the value of the *PS2_code*, as well as the value of a 1-bit flag concatenated to the *PS2_code*, which indicates whether the *PS2_code* is a make code or a break code (*PS2_make_code*), through the IORD function with the base address defined in Qsys and the offset register 0. If an IOWR is performed at the same base address and the same offset register, the *PS2_code_ready_irq* will be cleared. Note, this behavior is specific to our custom component.
You have to perform the following tasks in the lab:

- Get familiar with the hardware and software configuration of this design – it is to be noted that the PS2 component is integrated into the NIOS system through the Avalon bus and you do not need to make any changes to the hardware configuration in Qsys
- The interrupt service routine (ISR) for handling the PS2 codes manipulates global variables, which although it works, is not a good practice because global variables can be changed also by other functions; you are asked to change the ISR, in such way that it changes only the context, which is passed through a pointer to a data structure that contains the same information currently held in the following variables: `extern volatile char string_buffer[];` `extern volatile int cur_buf_length;` `extern volatile int buffer_flush;` Note, the type for this data structure should be defined in `"define.h"` however the variable(s) of this type should still be declared in the main function as `volatile` and the ISR should be registered with the pointer to the appropriate context

Experiment 3

In this experiment you will explore three different NIOS configurations (NIOS/e, NIOS/s and NIOS/f) and you will measure the performance of the bubble sorting algorithm using performance counters.

Processor type	NIOS/e	NIOS/s	NIOS/f
Included components	RISC 32-bit	RISC 32-bit Instruction Cache Branch Prediction Hardware Multiply Hardware Divide	RISC 32-bit Instruction Cache Branch Prediction Hardware Multiply Hardware Divide Barrel Shifter Data Cache Dynamic Branch Prediction

Table 1: Comparison of different NIOS processors.

Table 1 shows the important components in each of the three NIOS processor types. The simplest one, NIOS/e, contains only the 32-bit RISC processor with five-stage integer pipeline. NIOS/s extends NIOS/e with an instruction cache (the size can be configured by the user), a static branch predictor and hardware support for integer multiplication/division/shifting in three clock cycles (note, this latency can be user-configured). NIOS/f adds a dynamic branch predictor, a data cache and a barrel shifter (which is useful, for example, in encryption/decryption applications to perform rotates in a single clock cycle).

The performance of different NIOS configurations can be measured using performance counters. They are simple programmable counters with start/stop capability. Before they can be used, the component needs to be added to the NIOS system in Qsys. Subsequently the counters can be accessed using the functions defined in `"altera_avalon_performance_counter.h"`. There are a total of seven sections for each performance counter. All of them are reset using `PERF_RESET(PERFORMANCE_COUNTER_0_BASE);` where `PERFORMANCE_COUNTER_0_BASE` is the address as defined in Qsys. They can be enabled/disabled using `PERF_START_MEASURING(performance_name);` and `PERF_STOP_MEASURING(performance_name);` where `performance_name` is the address of the counter. Each section `k` can do the measurements for a code section marked by `PERF_BEGIN(performance_name, k);` and `PERF_END(performance_name, k);` statements.

There are three folders (*experiment3a*, *experiment3b* and *experiment3c*), which contain the same system configurations and the same software; the only difference lies in the type of the NIOS processor (NIOS/e, NIOS/s and NIOS/f respectively). You have to perform the following tasks in the lab:

- The performance counters are included in Qsys and the software is provided to access them in all the 3 folders; what is missing from the source code is the implementation for the `void bubble_sort(int *data_array, int size)` function; implement it and measure the performance of this simple sorting algorithm on the three NIOS systems and interpret the results

Write-up Template

COE4DS4 – Lab #3 Report
Group Number
Student names and IDs
McMaster email addresses
Date

There are 3 take-home exercises that you have to complete within one week. Label the projects as exercise1, exercise2 and exercise3. If, for any particular reason, you will add/remove/change the signals in the port list from the port names used in the design files from the in-lab experiments, make sure that these changes are properly documented in the source code.

Exercise 1 (4 marks) – Using the knowledge developed in *experiment1*, build a prototype embedded system on the DE2 board that emulates the digital part of an elevator, using the following input/output behavior and assumptions/constraints.

Inputs:

• SWITCH_I[17] is the default active low reset for the system;
• SWITCH_I[13] down to SWITCH_I[0] are used to indicate if a request has been made for the elevator in the corresponding floor; i.e., when SWITCH_I[*k*] is toggled (positive transition only), where *k* is from 0 to 13, a request is made that the elevator should stop at the k^{th} floor;
• PUSH_BUTTON_I[3] is used as “door open” button and its specific behavior is explained in the assumptions/constraints section;
• SWITCH_I[16], SWITCH_I[15], PUSH_BUTTON_I[2] and PUSH_BUTTON_I[1] are used to set up the amount of time the elevator spends between floors and at each floor while the doors are open – the specific behavior is explained in the assumptions/constraints section;
• SWITCH_I[14] and PUSH_BUTTON_I[0] are not used.

Outputs:

• RED_LED_O[13] down to RED_LED_O[0] will be lighted on when a request has been made at the corresponding floor and it should be turned off as soon as the elevator reaches the floor;
• The rightmost seven segment display should show the floor number where the elevator is located (in HEX format) and it should be displayed as soon as the elevator reaches the floor; note, if the elevator is moving up, then while the elevator is between floors *k* and *k+1*, the value to be displayed is *k+1*; similarly, if it is moving down, then while the elevator is between floors *k* and *k-1*, the value to be displayed is *k-1*.

Assumptions/constraints:

• You must use one custom peripheral attached to the NIOS processor that generates an interrupt whenever the elevator reaches a floor; the amount of time that it takes from one floor to another is programmable and it varies according to the values that are placed on SWITCH_I[16] and SWITCH_I[15] as follows: for 00 it takes 0.5 seconds, for 01 it takes 1 second, for 10 it takes 1.5 seconds and for 11 it takes 2 seconds. The value from SWITCH_I[16] and SWITCH_I[15] will only be loaded when PUSH_BUTTON_I[2] is pressed. It is important to note this timer should restart itself only when the elevator leaves a floor (*regardless* where it stops at that floor to service a request or not).
• You must use another custom peripheral attached to the NIOS processor that generates an interrupt to notify that the elevator should close its door. The amount of time that the door will stay open when the elevator reaches a floor is programmable and it varies according to the values that are placed on SWITCH_I[16] and SWITCH_I[15] as follows: for 00 it takes 0.5 seconds, for 01 it takes 1 second, for 10 it takes 1.5 seconds and for 11 it takes 2 seconds. The value from SWITCH_I[16] and SWITCH_I[15] will only be loaded when PUSH_BUTTON_I[1] is pressed; note, however after an elevator reaches the floor, even if the interrupt to close the door is generated, the door will not close while PUSH_BUTTON_I[3] is pressed. It is important to note this timer should restart itself only when the elevator arrives at a floor that

has been requested. Note also, if the timer has expired and PUSH_BUTTON_I[3] is pressed than the timer will not restart itself; rather, in this case, the door will be closed upon the release of PUSH_BUTTON_I[3];
• When the elevator is stationary (that is no requests are made and it stays at the last floor where it was asked to go to), it will stay as such until a request is made;
• To simplify the implementation, it is assumed that PUSH_BUTTON_I[2] and PUSH_BUTTON_I[1] are pressed only when the elevator is stationary; we assume that PUSH_BUTTON_I[3] is pressed only when the elevator is stopped at a requested floor and the door is already open or when the elevator is stationary;
• When the elevator is moving up it will service all the requests above the floor where the elevator is currently located, until it will change its direction; for example, if the elevator starts from floor 4 and the request for floor 8 is made, then it will start moving up; if while it moves up to floor 8, the request for floor 3 comes first and then the request for floor 9 comes next, the elevator will not change its direction until it completed its service to floor 9; conversely, when the elevator is moving down it will service all the requests below the floor where the elevator is currently located, until it will change its direction;
• The SWITCH_I and PUSH_BUTTON_I should pass their commands to the system through interrupts;
• It is ok to assume that while the ISR for SWITCH_I or PUSH_BUTTON_I is being executed, no other change in either SWITCH_I or PUSH_BUTTON_I will occur;
• You can interface the custom timers using either GPIO or custom components; or using a combination (of your choice) of GPIO and custom components.

Submit your sources and in your report write a quarter-of-a-page paragraph describing your reasoning.

Exercise 2 (1.5 marks) – *Extend* the in-lab contribution to *experiment2* to handle the following scenario. Only the caps lock and the character keys are supported. When caps lock is enabled the characters will be displayed (as per the rule explained below) in upper case; otherwise they are displayed in lower case. The caps lock mode is changed (i.e., toggled) when the caps lock key is pressed and then released. For the caps lock mode to toggle again, the caps lock key must be then pressed and released again. When a character key is pressed, the corresponding character is displayed. In order to be displayed again, the character key must be first released and then pressed again. If two or more character keys are pressed at the same time, the above rule should still be respected.

Submit your sources and in your report write a quarter-of-a-page paragraph describing your reasoning.

Exercise 3 (3.5 marks) –

In this exercise you will measure the effect of different system configurations (both software and hardware) on code size and code performance. This exercise has 3 parts:

Extend the in-lab code for experiment 3 to do the following: Generate an array of 500 random variables from 0 to 65535 (inclusive), then sort the array using your choice of sorting algorithm, then call the following function to add up some of the elements of this array, and then print the returned `sum` value.

```
int sum_function(int *data_array, int size)
{
	int i;
	int sum=0;
	for (i=0; i<size; i++)
		if (data_array[i] < 32768)
			sum += data_array[i];

	return sum;
}
```

Part (i) Effect of processor configuration:
Write a program that measures only the runtime of this function. Your measurement should not include the time required to print or sort. Your code should run 10 different measurement of 10 sets of randomly generated numbers and calculate the average of these measurements. Runtime clock cycles of the 10 measurements and the average must be clearly printed to the console.

Repeat the above for three different processor configurations provided (in the *experiment3a*, *experiment3b* and *experiment3c* folders). In your report include **only the average** measurements and **provide a concise interpretation** for the variation in performance (from one NIOS configuration to another).

Part (ii): Effect of compiler optimization:
You can choose the effort level of the compiler for optimizing your code (see Note below). GCC compiler built into the NIOS platform supports the following optimization levels: Optimization Off – Optimize Levels 0,1,2,3 – Optimize for code size.

Your task is to compare 3 different optimizations (Off / Level 3 / Size) for one of the processor configurations. In your report you should include: 1- Runtime of the `sum` function as explained above, with different optimization levels. 2- Code size in KB for each optimization level (see Note below). 3- Your interpretation of the results.

In order to choose your processor configuration, you should add the student number of the two group members, modulo 3, and choose for 0: *experiment3a*, for 1: *experiment3b* and for 2: *experiment3c*.

Part (iii): Effect of input data:
In part (i) the array was sorted before calling the sum function. Repeat the measurements of part(i) without sorting the array. For each configuration see if the sorted-ness of the array has any remarkable effect on the runtime. Provide concise interpretation for the variation in performance, if any.

Exclusively for this exercise: submit only a single source file named "exercise3.c" which is the code of part(i), regardless of the system configuration.

NOTE:

In order to change code optimization level in NIOS:
In project explorer right-click on experimentxx (without bsp), go to properties, from left window choose "Nios II Application Properties", then in the right window change the Optimization Level to the desired option. Choose "Size" or "Off" or "Level 3".

In order to check the executable code size in NIOS:
After a successful build of the project, in project explorer right-click on experimentxx (without bsp), go to "Nios II ->", select "Nios II Command Shell".
This will open a terminal in the same directory as your experiment's source codes and executable.
In the terminal enter:
```
nios2-stackreport experiment<xx>.elf
```

… where <xx> depends on the project's name. Type "exit" to close the shell.

VERY IMPORTANT NOTE:

This lab has a weight of 9% of your final grade. The report has no value without the source files, where requested. Your report must be in "pdf" format and together with the requested source files it should be included in a directory called "coe4ds4_group_xx_takehome3" (where xx is your group number). Archive this directory (in "zip" format) and upload it through Avenue to Learn before noon on the day you are scheduled for lab 4. Late submissions will be penalized.