COE4DS4 Lab #2
Introduction to Embedded Software for the NIOS II Processor

Objective

To get familiarized with the Altera's system integration flow (Qsys) and embedded processing using the NIOS II embedded processor and the Altera NIOS II embedded design suite.

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 tool flow for the following experiments. The tools that will be explored are the Qsys and NIOS II software development kit (SDK), as well as the links between them and the Quartus design environment and the DE2 board.

At the core of any embedded system is at least one embedded processor/microcontroller that manages the flow of data between the system's tasks. The peripherals, hardware accelerators, sensors/actuators, ..., are interfaced to the embedded processor, which controls them through device drivers written (for most of the cases) in assembly and/or C/C++. For the purpose of this course we will employ the DE2 board as the platform for embedded system prototyping. Because all the custom hardware blocks, including different versions of the embedded processor, are mapped onto the field-programmable gate array (FPGA), the entire system is referred to as an SOPC (system-on-a-programmable-chip). The tool flow is shown below in Figure 1. *In the lab directory there is a "readme_toolflow.txt" file that details each step.*

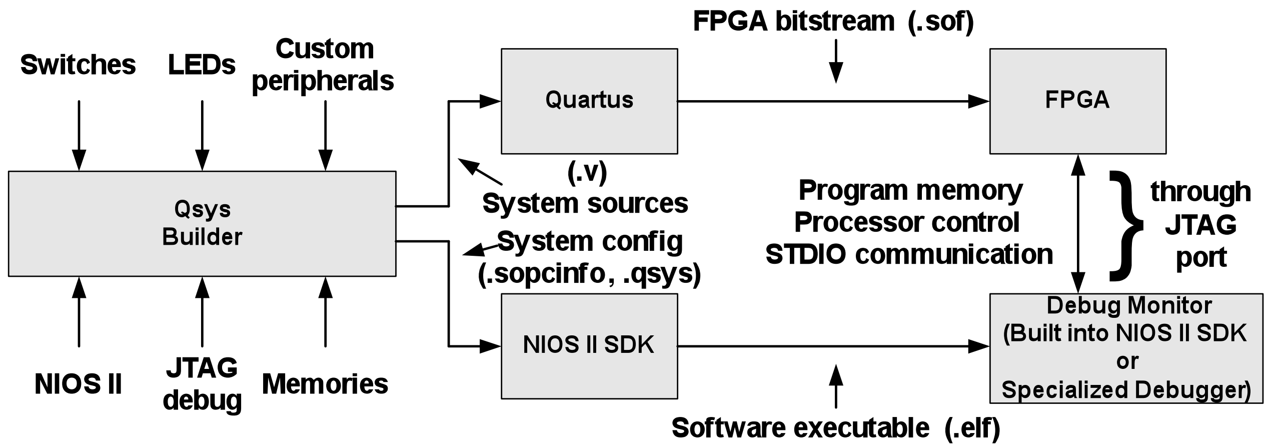

Figure 1: Embedded system development using Qsys, Quartus and NIOS SDK.

In our experiments, at the core of the embedded system is the NIOS II embedded processor available from Altera (**note**: NIOS II and NIOS will be used interchangeably for the rest of this course). NIOS comes as a *soft core*, which is in a proprietary format that can be integrated as a black box in the design (as a component instantiation). The hardware blocks (peripherals, hardware accelerators, ...) are written in Verilog and all of them glued together with the NIOS processor in a top-level Verilog file. This top-level module is generated through the Qsys, which captures the dependencies between the NIOS and the hardware blocks. Thereafter, the top-level Verilog file is compiled in Quartus to generate the SRAM object file (.sof), which is downloaded onto the FPGA. Once the FPGA has been configured, the NIOS SDK is used to write the software that run on the NIOS processor embedded into the FPGA. After the program is compiled in the executable and linkable format (.elf), the debug monitor (either part of NIOS SDK, as it is our case, or a specialized debugger) uploads the program onto the FPGA board and controls its execution through a debug port accessible through a terminal on the Host machine (called Host terminal).

The NIOS processor is a 32-bit microprocessor that can be configured in different modes. In the simplest mode that consumes less than 1,000 logic elements (LEs) in the FPGA, the performance of the microprocessor is up to 5 DMIPS (Dhrystone Millions of Instructions Per Second) at a 50 MHz operating frequency. It has 32 integer registers (32-bits each) and it can perform the basic arithmetic, logic, relational and shift operations. In more advanced configurations, dedicated hardware for multiplication/division can be included, as well as branch prediction and caching. The DMIPS performance can be improved by a factor of 5 at the expense of doubling the resource usage. In the most advanced configuration, the performance can be further enhanced with barrel shifters, dynamic branch predictors and data caches.

Caches and memories attached to NIOS can be configured in the Qsys, as well as the levels of the debug module. Each of these will consume more memory/LEs in the FPGA. For example, level 1 debugging that supports only target connection, software download and software breakpoints will use approx 300 LEs; the most advanced debug level with hardware breakpoints, data triggers, instruction/data traces, off-chip traces will use approx 3,000 LEs. Custom instructions, such as supporting floating-point hardware (single-precision IEEE-754) or bit swapping will consume additional FPGA resources. User-defined instructions can also be included, which can be used to accelerate the application at hand.

The NIOS processor communicates with its peripherals through the *Avalon* bus. The transactions on this bus are on 32, 16 or 8 bits at a time between a bus master (e.g., NIOS processor) and a slave (peripheral/memory). As soon as a transaction is complete, the bus is available in the following clock cycle. In the case that multiple masters are requesting data from the same slave, there will be slave-side arbitration which decides which master accesses the slave. This enables multiple masters to carry out transactions on the same bus simultaneously (so long as they do not compete for the same slave). The Avalon instruction bus is used for transferring data to/from memories, while the Avalon data bus is used for transferring data to/from any type of peripherals ranging from pushbuttons to networking controllers.

Downloading the software programs and debugging is performed through the Joint Test Action Group (JTAG) port on the FPGA. The same JTAG port can be used for monitoring the NIOS registers in the FPGA, controlling the program execution and collection of trace data; as well as for handling the basic STDIO to the Host terminal. To achieve the communication with the Host terminal, the JTAG UART port needs to be included as part of the system built in Qsys. Figure 2 shows the NIOS processor with 1.5 Kbytes of memory and the JTAG UART peripheral implemented in experiment 1.

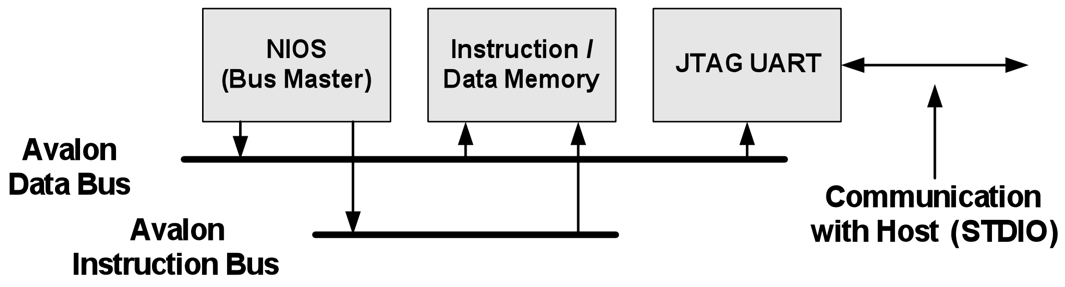

Figure 2: The basic configuration of the NIOS processor used to communicate with the Host.

You have to perform the following tasks in the lab:

- You will learn the tool flow from Figure 1 by building the system from Figure 2 for printing "Hello from NIOS II!" on the Host terminal. You will use the Altera built-in types for integers (`alt_u16` for unsigned on 16 bits or `alt_16` for signed) and printing (`alt_putstr`). Note, to use the integers you need to include the following library: `#include "alt_types.h"`
- The program will be subsequently modified to compute and print the first 5 powers of 2 in hex format.

Experiment 2

The aim of this experiment is to show you how the peripherals from the DE2 board can be connected to the NIOS processor. You will be given a reference design for Qsys where the 18 input switches are grouped as one general purpose I/O (GPIO); the LEDs and the 7-segment displays are also configured as GPIOs (see Figure 3). Communicating with GPIO devices is done through the IORD and IOWR functions, where the name of the peripheral is the first argument, the base address is the second and the third is the write value. The base address is set up in Qsys and when the libraries are built in the NIOS II SDK, the "system.h" will be generated and it will contain the defines of all the offsets.

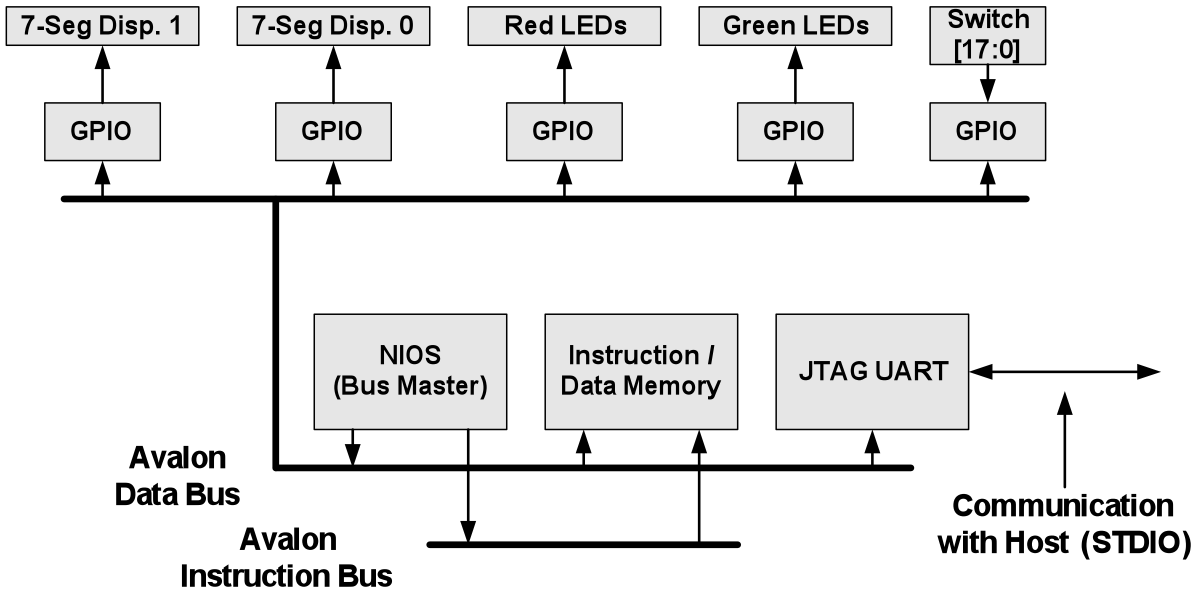

Figure 3: The GPIOs from the DE2 board connected to the NIOS processor.

For GPIO peripherals, there are four offsets added to the base address, which are used to access the registers in the memory space of the peripheral: at offset 0 is the data register which can be read or written; at offset 1 is the direction register (used with three-stated I/Os; not applicable to this lab); at offset 2 is the interrupt mask register (see experiment3b for its usage); at offset 3 is the edge detect register where each time a positive/negative edge occurs (depending on the setting in Qsys), the register will be set (this is an important feature for interrupt handling, as explained also in experiment3b); the resetting of offset register 3 is the user's duty as part of his interrupt service routine.

You have to perform the following tasks in the lab:

- You are given the code that polls the switches and it displays the position of the most significant switch (that is turned on) on the two rightmost 7-segment displays. You are asked to change the code to display the position of the least significant switch (that is turned on).
- Add the code to drive the two rightmost green LEDs with the AND between the SWITCHES 17 and 16 and the OR between SWITCHES 1 and 0 respectively.

Experiment 3

Part (a) The aim of this experiment is to introduce the character LCD. The system includes all the peripherals from experiment 2 plus the character LCD. You have to perform the following tasks in the lab:

- Copy the "C" code from the main function from experiment 2 which interfaces to the switches, LEDs and seven segment displays. Whenever the position of a switch is changed, the following message should be displayed on the character LCD: "Switches changed".

Part (b) The aim of this experiment is to introduce interrupts and understand how to use them. We will monitor changes on the input switches and print a message in the Host terminal, whenever a change occurs. Unlike the previous experiment, the peripherals will interact with the processor through interrupts (as shown in Figure 4). Also, the switches are broken into two separate groups (Groups A and B) and each of these groups can generate its own interrupts.

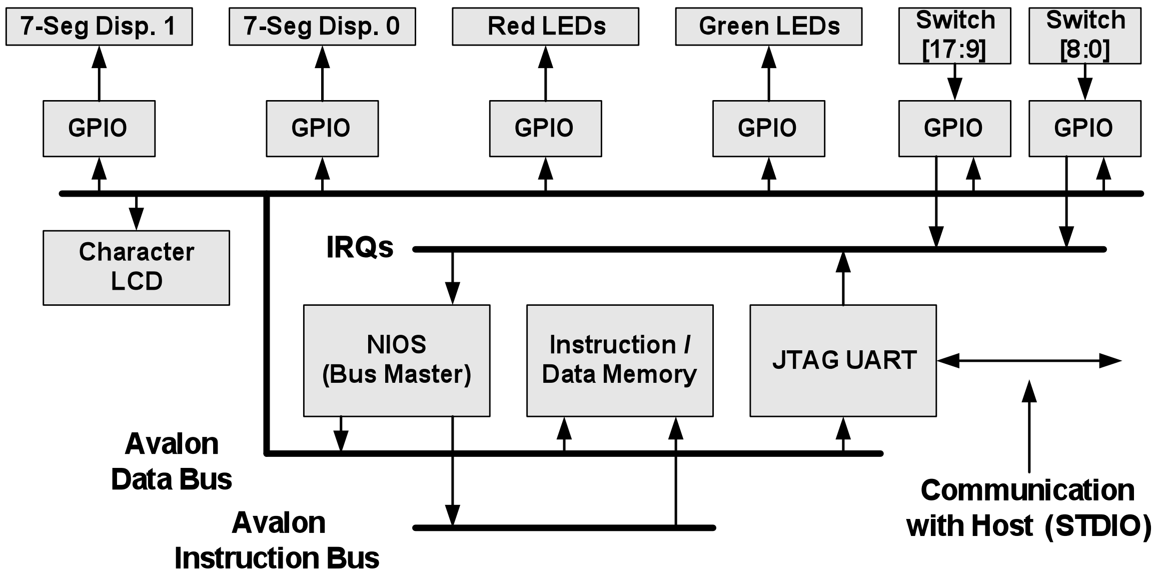

Figure 4: Interfacing interrupts from the GPIO to the NIOS processor.

Three main modifications to the given code are necessary to set up the interrupts. First, include the header file for the interrupt library by adding the code `#include "sys/alt_irq.h"` at the top of the C source file. Next, create the interrupt service routine (ISR) which is the function that will be called on interrupt. Consider the following very simple ISR:

```
void SW_GRPA_interrupt(void) {
    alt_printf("Switches GRPA changed\n");
    IOWR(SWITCH_GRPA_I_BASE, 3, 0x0);
}
```

This ISR will print a message to the terminal each time any of the switches 0 to 8 (Group A) are changed. The purpose of the IOWR is to clear the edge detection register, which keeps the ISR from being called multiple times for the same event. Finally the interrupt must be initialized using the following code:

```
IOWR(SWITCH_GRPA_I_BASE, 3, 0x0); // edge capture register
IOWR(SWITCH_GRPA_I_BASE, 2, 0x1FF); // IRQ mask
alt_irq_register(SWITCH_GRPA_I_IRQ, NULL, (void*)SW_GRPA_interrupt);
```

The first line initializes the edge detection register to zero. An interrupt will occur each time the edge detection register is different than 0. The second line enables an interrupt on the change of any of the 9 switches by setting up the IRQ mask register to all ones. If any of the switches are to be ignored then the mask register must be re-programmed (e.g., to ignore the two least significant switches write 0x1FC).The third line installs the interrupt by linking the interrupt to the peripheral with the ISR we have written.

You have to perform the following tasks in the lab:

- Implement the interrupts for Group A by making the changes to the source code as detailed above
- Enable the interrupts only for changes on Switch 0 and display “Switch 0 changed” on the character LCD

Write-up Template

COE4DS4 – Lab #2 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 (2.5 marks) – Change the in-lab *experiment2* to drive the green LEDs as follows. Positions 8, 7, 6 and 5 of the green LEDs will be lightened ("on") only if number of switches that are in the high position ("on") is strictly greater than the number of switches that are in the down position ("off"); otherwise these four green LEDs will not be lightened ("off"). The five rightmost green LEDs (positions 4, 3, 2, 1 and 0) will be controlled as follows. If SWITCH[17] is "on" then, if at least one of the remaining switches is "off", display the binary representation of the most significant switch that is "off"; for example, if the most significant switch that is "off" is 13, then the green LEDs 3, 2 and 0 will be "on" and green LEDs 4 and 1 will be "off". If SWITCH[17] is "on" and all the other switches are "on" then all the five rightmost green LEDs will be "on". If SWITCH[17] is "off" and if SWITCH[16] is "on" then display the binary representation of the least significant switch that is "off". If both SWITCH[17] and SWITCH[16] are "off" then display the binary representation for the number of switches that are "on".

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

Exercise 2 (2.5 marks) – Extend the *experiment3a* to monitor SWITCHES [15:0] in order to record the most recent 16 values obtained after toggling any of these 16 switches (note, on power-up the 16 recorded values are assumed to be all zeros). The binary combinations of SWITCHES [15:0] are interpreted as signed decimals on 16 bits.

Each time when SWITCH[16] changes from low to high, print (in decimal format) on the character LCD the two largest of the previous 16 recorded values (one value per row, which must be right justified). The character LCD does not need to change until a new rising edge on SWITCH[16] is detected. Similarly, when SWITCH[16] changes from high to low, print on the terminal two separate lines as follows: in the first line print (also in decimal format) the entire sequence of the 16 values that were recorded when any of SWITCHES [15:0] was toggled; in the following line print the longest strictly monotonic decreasing subsequence from the sequence printed on the previous line. A subsequence `S` with `N` elements is strictly monotonic decreasing if for any `i` from `0` to `N-2` we have `S[i]` strictly greater (>) than `S[i+1]`. If there are two or more strictly monotonic decreasing subsequences of the same length, then the one whose last element was recorded most recently should be printed. If no sub-sequence exists, then instead of printing the sub-sequence you should print a message that confirms the requested sub-sequence does not exist.

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

Exercise 3 (3 marks) – Consider the embedded system setup from *experiment3b*, which can be used to prototype a bus ticket machine, as explained below. You are on McMaster campus and you want to take the bus to Toronto, Ottawa or Montreal. The four least significant bits of Group A are connected to input switches, which are used to feed coins to the ticket machine. Whenever they toggle (either positive or negative edge) the amount accumulated for the current transaction is updated as follows:

• Position 0 of Group A – 1 dollars
• Position 1 of Group A – 2 dollars
• Position 2 of Group A – 5 dollars
• Position 3 of Group A – 10 dollar

The four least significant bits of Group B are connected to four input switches, which give the commands to the ticket machine as follows:

• Position 0 of Group B – purchase ticket to Montreal: the price of the ticket is 41 dollars
• Position 1 of Group B – purchase ticket to Ottawa: the price of the ticket is 32 dollars
• Position 2 of Group B – purchase ticket to Toronto: the price of the ticket is 15 dollars
• Position 3 of Group B – cancel the current transaction

The four least significant bits of the output peripheral Green LEDs are showing the following:

• Position 0 of Green LED – activated so long as there is an active transaction in progress
• Position 1 of Green LED – activated for 2.5 seconds only (after a transaction completes with a cancel)
• Position 2 of Green LED – activated for 2 second only (after a transaction completes with a purchase)
• Position 3 of Green LED – activated for 1.5 seconds only when there is no sufficient amount for the current transaction - for pausing the NIOS processor for 1.5 seconds, you can use the `usleep(1500000)` (note, you will need to `#include <unistd.h>` in order to use `usleep` the function)

In addition to the behavior described above, we have the following simplifying assumptions:

• To simplify the interrupt handling, it is assumed that no two (or more) inputs will ever change at the same time (note, it does not matter if the inputs are from the same group or not); hence both interrupt service; routines will work under the assumption that only one bit has been changed in the edge register;

• While an interrupt is serviced, it is assumed that no other interrupt request will come (note, it does not matter whether the serviced interrupts or the interrupt requests are from Group A or B);

• The machine will never overflow (neither in the number of coins nor for the representation of amount);

• It is assumed that no interrupt occurs before the program execution enters the `while (1);` loop;

• A new transaction becomes active only if the amount is 0 and any of the Group A switches is toggled;

• If a purchase or cancel command is given when a transaction is not active then no action will be taken;

• During an active transaction, the amount is updated (using the values dependent on the switch positions, as described above) when any of the Group A switches is toggled (either edge will trigger the interrupt);

• If a purchase command has been given for an active transaction when there is no sufficient amount in the ticket machine then output LED from position 3 will be activated for 1.5 seconds (described above); note however, the current transaction will continue with the same amount as before the command was given;

• A ticket has been purchased if a purchase command was given for a particular ticket and the amount accumulated for the current transaction is greater than or equal to the price of the respective destination;

• A transaction completes either when the cancel command is given or when a ticket was purchased (and the amount becomes 0 upon transaction completion); note, if there is any change to be provided, then its value should be displayed (in decimal) on the two rightmost seven segment displays (it is assumed the change does not exceed the maximum value that can be displayed); the change is displayed for as long as the corresponding green LED is on (see above for position and duration); at any other time the seven segment displays are not lighted;

• The interrupt services will operate on the context that is passed, which is the amount of money available for the transaction in progress. There should be two interrupt service routines to be written by you:
`void SW_GRPA_interrupt(int *amount) {…} and void SW_GRPB_interrupt(int *amount) {…}.`

The following main code must be used and should NOT be modified.

```
int main(void)
{
    volatile int amount = 0;

    IOWR(SWITCH_GRPA_I_BASE, 3, 0x0);
    IOWR(SWITCH_GRPA_I_BASE, 2, 0xF);
    alt_irq_register(SWITCH_GRPA_I_IRQ, &amount, (void*)SW_GRPA_interrupt);

    IOWR(SWITCH_GRPB_I_BASE, 3, 0x0);
    IOWR(SWITCH_GRPB_I_BASE, 2, 0xF);
    alt_irq_register(SWITCH_GRPB_I_IRQ, &amount, (void*)SW_GRPB_interrupt);

    IOWR(LED_GREEN_O_BASE, 0, 0x0);
    while (1);
    return 0;
}
```

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

VERY IMPORTANT NOTE:

This lab has a weight of 8% 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_takehome2" (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 3. Late submissions will be penalized.