Imperial College
London

DSD

COURSEWORK

v3.2 – Christos Bouganis

TABLE OF CONTENTS

Requirements to complete this coursework:

- Quartus II (v.12);
- Nios II Embedded Design Suite (v.12);
- Modelsim (Modelsim-Altera or ActiveHDL);
- Matlab (or Python);
- Terasic DE0 board;
- Coursework source files including DE0 documentation;
- On-line references from Altera.

Legend:

ⓘ Important information.

📖 Reference material that you should read before proceeding.

🛠 Work you should deliver. A set of tasks for you to do, and deliverables to include in your report.

🕷 Debug hints

☺ Helpful information

Please read this document carefully before starting the coursework as it will guide you to complete the coursework efficiently. There are links to references that explain, and exemplify, procedures you will need to know how to do. You're advised to read and understand them.

INTRODUCTION

In this coursework you will acquire experience in the design of digital systems. As a case study, we will focus on the acceleration of computationally intensive applications and we will see how having customising hardware blocks tuned for a certain application can lead to a better performing system compared to one that does not have these computational units.

In this coursework we will start by running applications on a generic central processing unit (CPU), assess the system's performance and then focus on the design of dedicated hardware processing elements for the acceleration of the same applications.

The system design will target a Field-Programmable Gate Array (FPGA) from Altera. They are generic digital devices that support the implementation of any digital hardware design after their fabrication, including the implementation of soft-processors. The FPGA you're going to use is on a development kit (DE0) from Terasic, which is available in the lab. The target CPU is the soft-processor NIOS II, also provided by Altera. More information about NIOS II can be found here:

http://www.altera.com/devices/processor/nios2/ni2-index.html.

The main application for configuration of the FPGA is Quartus II from Altera. It allows to design and program digital systems based on FPGAs. The Qsys tool allows the implementation of a complex digital system including a CPU and all the required components to make it work. You can find more about it here:

http://www.altera.com/literature/hb/qts/qsys_intro.pdf .

You will program the NIOS II processor by writing C code using EDS. How to do this is described in the tutorial pointed out in the hand-out.

In the coursework you are about to do, you'll start by describing your system in Qsys (which you can access from Quartus II), synthesize in Quartus, and then configure the FPGA. At this point you don't need to write any Verilog/VHDL. Verilog is a hardware description language. Later you can use verilog /block diagrams/VHDL to describe your customized hardware blocks.

Objective: Acceleration of Arithmetic Functions

In this coursework you are going to see how dedicated hardware can be applied to a general purpose digital system in order to accelerate certain parts of the calculations. The more specialised the hardware blocks are, the faster the computation becomes, but that's reflected in FPGA resources or circuit area.

The main purpose of this coursework is to create a digital system that evaluates the following function:

$$f(x) = \sum_{i=1}^{N} 0.5 * x_i + x_i^2 \cos\left(floor\left(\frac{x_i}{4}\right) - 32\right) \quad (1)$$

for **<u>any</u>** given input vector **x**, where each element of the vector is a single-precision floating-point number.

In order to test and assess the performance of your system, you should use the following test cases (Matlab notation):

1. X=0:5:255
2. X =0:0.1:255
3. X = 0:0.001:255

where x is a single-precision floating-point number. For example, the first case describes an input vector X with the following elements: [0, 5, 10, 15, …, 255].

Your design should be assessed under the following metrics (when they are applicable).

- FPGA resources used by your system (Logic Elements, Embedded Multipliers, Memory Bits). As the target device is fixed, the proposed metric for FPGA resources is the weighted sum of the utilised resources. For example, if your design uses *LE* Logic Elements, *EM* Embedded Multipliers, and *MB* Memory Bits, then the FPGA resources are:

$$\text{FPGA resources} = \frac{1}{3}\left(\frac{LE}{Total\ LE\ in\ the\ device} + \frac{EM}{Total\ EM\ in\ the\ device} + \frac{MB}{Total\ MB\ in\ the\ device}\right)$$

- Latency or Execution time – This is the time that is needed for your system to compute (1) for an input vector X. This includes only the time needed for the evaluation of (1) for the whole vector, without taking into account any initialisation time and so on.

- Throughput – Number of results per second.

- Application size (program).

Thus, you should be able to plot the performance of your design in a performance vs FPGA resources plot and compare different solutions.

Please note that at the end of the coursework you need to provide the code for each Task. Please check the coursework submission details for more information on this to avoid any duplication of work.

Task 1: NIOS II Setup

In this task you will get familiar with the design of a NIOS II processor and the design choices that impact the system's performance.

Altera, the manufacturer of your target FPGA, offers an embedded processor (also known as soft-processor), named NIOS II, which can be instantiated in any design, and even more than once. The software programming for NIOS II is done in EDS (Eclipse) using C code.

Learn how to setup a system with a NIOS II processor, and start building your project. Download the project template (system_template_de0) available on the DSD coursework page. Place the files to a directory of your choice.

Do not use spaces in the directory path name

Do not download and install the design files from Altera

Your main file is the hello_word.qpf. Open the file to see what has been given to you. You will see just some input/output pins only. These pins are connected to specific pins on the FPGA of your board in order to allow your design to communicate with the outside world. In the Project Navigator you will see that the Cyclone III device has been selected as the target device. This is the device that comes with your DE0 board.

The rest of the section is a summary extracted by the NIOS II tutorial provided by Altera (http://www.altera.com/literature/tt/tt_nios2_hardware_tutorial.pdf).

Create a New Qsys System

The design of the processor is a complex process, but Altera provides a tool (Qsys) to help you with this task. Through Qsys, you can generate a NIOS II processor system, add the desired components, and configure how they connect together.

Click Qsys on the Tools menu in the Quartus II software. A new window will open and your system will have on the clock signal.

Define the System in Qsys

You use Qsys to define the hardware characteristics of the Nios II system, such as which Nios II core to use, and what components to include in the system. Qsys does not define software behavior, such as where in memory to store instructions or where to send the stderr character stream.

In this section, you perform the following steps:

1. Specify target FPGA and clock settings.

2. Add the Nios II core, on-chip memory, and other components.

3. Specify base addresses and interrupt request (IRQ) priorities.

4. Generate the Qsys system.

The Qsys design process does not need to be linear. The design steps in this tutorial are presented in the most straightforward order for a new user to understand. However, you can perform Qsys design steps in a different order.

Specify Target FPGA and Clock Settings

The **Clock Settings** and the **Project Settings** tabs specify the Qsys system's relationship to other devices in the system.

Perform the following steps:

1. On the **Project Settings** tab, select the **Device Family** that matches the Altera FPGA you are targeting. If a warning appears stating the selected device family does not match the Quartus project settings, ignore the warning. You specify the device in the Quartus project settings later in this tutorial. Since you have run Qsys from your project, where the device has been already specified, these fields should be already populated.

2. In the documentation for your board look up the clock frequency of the oscillator that drives the FPGA.

3. Got to **View -> Clocks** to open the **Clocks** tab, double-click the clock frequency in the **MHz** column for clk_0. clk_0 is the default clock input name for the Qsys system. The frequency you specify for clk_0 must match the oscillator that drives the FPGA. Your board has a 50MHz oscillator. Set this value, if it has not been already populated.

Next, you begin to add hardware components to the Qsys system. As you add each component, you configure it appropriately to match the design specifications.

Add the On-Chip Memory

Processor systems require at least one memory for data and instructions. As a starting point, let's use one 20 KB on-chip memory for both data and instructions. To add the memory, perform the following steps:

1. On the **Component Library** tab (to the left of the **System Contents** tab), expand **Memories and Memory Controllers**, expand **On-Chip**, and then click **On-Chip Memory (RAM or ROM).**

2. Click **Add**. The On-Chip Memory (RAM or ROM) parameter editor appears.

3. In the **Block type** list, select **Auto**.

4. In the **Total memory size** box, type 20480 to specify a memory size of 20 KB.

Do not change any of the other default settings

5. Click **Finish**. You return to Qsys.

6. Click the **System Contents** tab. An instance of the on-chip memory appears in the system contents table.

7. In the **Name** column of the system contents table, right-click the on-chip memory and click **Rename**.

8. Type onchip_mem and press Enter.

In the System Contents you can see that the memory has been added and possible connections to existing signals have been highlighted. We will come back to these connections later.

Add the Nios II Processor Core

In this section you add the Nios II/s core and configure it to use 2 KB of on-chip instruction cache memory. For educational purposes, the tutorial design example uses the Nios II/s (standard) core, which provides a balanced trade-off between performance and resource utilization. In reality, the Nios II/s core is more powerful than necessary for most simple control applications.

Perform the following steps to add a Nios II/s core to the system:

1. On the **Component Library** tab, expand **Processors**, and then click **Nios II Processor**.

2. Click **Add**. The Nios II Processor parameter editor appears, displaying the **Core Nios II** tab. A new GUI pops-up.

3. Under **Select a Nios II core**, select **Nios II/s**.

4. In the **Hardware multiplication type** list, select **None**.

5. Turn off **Hardware divide**.

6. Click **Finish**. You return to the **Qsys System Contents** tab, and an instance of the Nios II core appears in the system contents table. Ignore the exception and reset vector error messages. You resolve these errors in steps 13 and 14.

7. In the **Name** column, right-click the Nios II processor and click **Rename**.

8. Type cpu and press Enter.

9. In the **Connections** column, connect the clk port of the **clk_0** clock source to both the clk1 port of the on-chip memory and the clk port of the Nios II processor by clicking the hollow dots on the connection line. The dots become solid indicating the ports are connected.

10. Connect the clk_reset port of the **clk_0** clock source to both the reset1 port of the on-chip memory and the reset_n port of the Nios II processor.

11. Connect the s1 port of the on-chip memory to both the data_master port and instruction_master port of the Nios II processor.

This last step connects the NIOS processor with the on-chip memory that you have instantiated through the Avalon bus. The NIOS processor serves as the master in this connection (as it initiates transactions) where the memory has a slave interface. The jtag signals are used for debugging purposes.

12. Double-click the Nios II processor row of the system contents table to reopen the Nios II Processor parameter editor.

13. Under **Reset Vector**, select **onchip_mem.s1** in the **Reset vector memory** list and type 0x0 in the **Reset vector offset** box. This tells to the processor where to reset its Program Counter (PC) in case of a reset.

14. Under **Exception Vector**, select **onchip_mem.s1** in the **Exception vector memory** list and type 0x20 in the **Exception vector offset** box.

15. Click the **Caches and Memory Interfaces** tab.

16. In the **Instruction cache** list, select **2 Kbytes**.

17. In the **Burst transfers** list, select **Disable**.

18. In the **Number of tightly coupled instruction master port(s)** list, select **None**.

Do not change any settings on the **Advanced Features, MMU and MPU Settings, JTAG Debug Module,** or **Custom Instruction** tabs.

19. Click **Finish**. You return to the Qsys **System Contents** tab.

Add the JTAG UART

The JTAG UART provides a convenient way to communicate character data with the Nios II processor through the USB-Blaster download cable. Perform the following steps to add the JTAG UART:

1. On the Component Library tab, expand Interface Protocols, expand Serial, and then click JTAG UART.

2. Click Add. The JTAG UART parameter editor appears. Do not make any changes.

3. Click **Finish**. You return to the Qsys **System Contents** tab, and an instance of the JTAG UART appears in the system contents table.

4. In the **Name** column, right-click the JTAG UART and click **Rename**.

5. Type jtag_uart and press Enter.

6. Connect the clk port of the **clk_0** clock source to the clk port of the JTAG UART.

7. Connect the clk_reset port of the clk_0 clock source to the reset port of the JTAG UART.

8. Connect the data_master port of the Nios II processor to the avalan_jtag_slave port of the JTAG UART.

Please note that the instruction_master port of the Nios II processor does not connect to the JTAG UART because the JTAG UART is not a memory device and cannot send instructions to the Nios II processor.

Add the Interval Timer

Most control systems use a timer component to enable precise calculation of time. To provide a periodic system clock tick, the Nios II HAL requires a timer. Perform the following steps to add the timer:

1. On the **Component Library** tab, expand **Peripherals**, expand **Microcontroller Peripherals**, and then click **Interval Timer**.

2. Click **Add**. The Interval Timer parameter editor appears.

3. In the **Presets** list (on the right), select **Full-featured**. Please make a note of the Timeout period setting.

4. Click **Finish**. You return to the Qsys **System Contents** tab, and an instance of the interval timer appears in the system contents table.

5. In the **Name** column, right-click the interval timer and click **Rename**.

6. Type sys_clk_timer and press Enter.

7. Connect the clk port of the **clk_0** clock source to the clk port of the interval timer.

8. Connect the clk_reset port of the **clk_0** clock source to the reset port of the interval timer.

9. Connect the data_master port of the Nios II processor to the s1 port of the interval timer.

Add the System ID Peripheral

The system ID peripheral safeguards against accidentally downloading software compiled for a different Nios II system. If the system includes the system ID peripheral, the Nios II SBT for Eclipse can prevent you from downloading programs compiled for a different system.

Perform the following steps to add the system ID peripheral:

1. On the **Component Library** tab, expand **Peripherals**, expand **Debug and Performance**, and then click **System ID Peripheral**.

2. Click **Add**. The System ID Peripheral parameter editor appears.

3. Click **Finish**. You return to the Qsys **System Contents** tab, and an instance of the system ID peripheral appears in the system contents table.

4. In the **Name** column, right-click the system ID peripheral and click **Rename**.

5. Type sysid and press Enter.

6. Connect the clk port of the **clk_0** clock source to the clk port of the system ID peripheral.

7. Connect the clk_reset port of the **clk_0** clock source to the reset port of the system ID peripheral.

8. Connect the data_master port of the Nios II processor to the control_slave port of the system ID peripheral.

Add the PIO

PIO signals provide an easy method for Nios II processor systems to receive input stimuli and drive output signals. Complex control applications might use hundreds of PIO signals which the Nios II processor can monitor. This design example uses eight PIO signals to drive LEDs on the board.

Perform the following steps to add the PIO.

1. On the **Component Library** tab, expand **Peripherals**, expand **Microcontroller Peripherals**, and then click **PIO (Parallel I/O)**.

2. Click **Add**. The PIO (Parallel I/O) parameter editor appears.

3. Do not change the default settings. Click Finish and return to Qsys. In the name column right-click the PIO and click rename. Give the name **led_pio** and press enter.

4. Connect the **clk** port to **clk_0** clock source, connect the **reset** port to **clk_reset** of clk_0, and connect the **data_master** port of the NIOS II processor to the **s1** port of PIO

5. In the external connection row, click **Click to export** in the **Export** column to export the PIO ports.

Specify Base Addresses and Interrupt Request Priorities

At this point, you have added all the necessary hardware components to the system. Now you must specify how the components interact to form a system. In this section, you assign base addresses for each slave component, and assign interrupt request (IRQ) priorities for the JTAG UART and the interval timer. Qsys provides the **Assign Base Addresses** command which makes assigning component base addresses easy. For many systems, including this design example, **Assign Base Addresses** is adequate. However, you can adjust the base addresses to suit your needs. Below are some guidelines for assigning base addresses:

■ Nios II processor cores can address a 31-bit address span. You must assign base address between 0x00000000 and 0x7FFFFFFF.

■ Nios II programs use symbolic constants to refer to addresses. Do not worry about choosing address values that are easy to remember.

■ Address values that differentiate components with only a one-bit address difference produce more efficient hardware. Do not worry about compacting all base addresses into the smallest possible address range, because this can create less efficient hardware.

■ Qsys does not attempt to align separate memory components in a contiguous memory range. For example, if you want an on-chip RAM and an off-chip RAM to be addressable as one contiguous memory range, you must explicitly assign base addresses.

Qsys also provides an **Assign Interrupt Numbers** command which connects IRQ signals to produce valid hardware results. However, assigning IRQs effectively requires an understanding of how software responds to them. Because Qsys does not know the software behavior, Qsys cannot make educated guesses about the best IRQ assignment.

The Nios II HAL interprets low IRQ values as higher priority. The timer component must have the highest IRQ priority to maintain the accuracy of the system clock tick.

To assign appropriate base addresses and IRQs, perform the following steps:

1. On the System menu, click **Assign Base Addresses** to make Qsys assign functional base addresses to each component in the system. Values in the **Base** and **End** columns might change, reflecting the addresses that Qsys reassigned.

2. In the **IRQ** column, connect the Nios II processor to the JTAG UART and interval timer.

3. Click the IRQ value for the **jtag_uart** component to select it.

4. Type 16 and press Enter to assign a new IRQ value.

5. Click the IRQ value for the **sys_clk_timer** component to select it.

6. Type 1 and press Enter to assign a new IRQ value.

Generate the Qsys System

You are now ready to generate the Qsys system. Perform the following steps:

1. Click the **Generation** tab.

2. Select **None** in both the **Create simulation model** and **Create testbench Qsys system** lists. Because this tutorial does not cover the hardware simulation flow, you can select these settings to shorten generation time.

3. Click **Generate**. The **Save changes?** dialog box appears, prompting you to save your design.

4. Type first_nios2_system in the **File name** box and click **Save**. The **Generate** dialog box appears and system generation process begins. The generation process can take several minutes. Output messages appear as generation progresses. When generation completes, the final "Info: Finished: Create HDL design files for synthesis" message appears.

5. Click **Close** to close the dialog box.

6. On the File menu, click **Exit** to close Qsys and return to the Quartus II software.

Congratulations! You have finished creating the Nios II processor system. You are ready to integrate the system into the Quartus II hardware project and use the Nios II SBT for Eclipse to develop software.

Integrate the Qsys System into the Quartus II Project

To instantiate the system module in the .bdf (i.e. your schematic capture file), perform the following steps (by closing Qsys, you should have gone back to Quartus on your hello_world system):

1. Double-click in the empty space to the right of the input and output wires. The Symbol dialog box appears.

2. Under Libraries, expand Project.

3. Click first_nios2_system. The Symbol dialog box displays the first_nios2_system symbol.

4. Click OK. You return to the .bdf schematic. The first_nios2_system symbol tracks with your mouse pointer.

5. Position the symbol so the pins on the symbol align with the wires on the schematic.

6. Click to anchor the symbol in place.

7. Make the connections:

    a) connect clk_clk to iCLK_50

    b) connect reset_rest_n to Vcc (you need to get this from the Quartus library)

    c) connect the leds.

    d) leave the DRAM pins unconnected.

7. To save the completed .bdf, click Save on the File menu.

Compile the Quartus II Project and Verify Timing

At this point you are ready to compile the Quartus II project and verify that the resulting design meets timing requirements. You must compile the hardware design to create a .sof that you can download to the board. After the compilation completes, you must analyze the timing performance of the FPGA design to verify that the design will work in hardware. In this case, the sdc file (i.e. the Synopsys Design Constraints File) has already been included to your project (by us), so you do not have to worry about this. You just need to know that the target clock frequency for the design is 50MHz.

To compile the Quartus II project, perform the following steps:

1. On the Processing menu, click **Start Compilation**. The Tasks window and percentage and time counters in the lower-right corner display progress. The compilation process can take several minutes. When compilation completes, a dialog box displays the message "Full Compilation was successful."

2. Click **OK**.

3. Expand the **TimeQuest Timing Analyzer** category in the compilation report.

4. Click **Multicorner Timing Analysis Summary**.

5. Verify that the **Worst-case Slack** values are positive numbers for **Setup**, **Hold**, **Recovery**, and **Removal**. If any of these values are negative, the design might not operate properly in hardware. To meet timing, adjust Quartus II assignments to optimize fitting, or reduce the oscillator frequency driving the FPGA. (You should not have to worry for any of the above).

Congratulations! You have finished integrating the Nios II system into the Quartus II project. You are ready to download the **.sof** to the target board.

Download the Hardware Design to the Target FPGA

In this section you download the **.sof** to the target board. Perform the following steps:

1. Connect the board to the host computer with the download cable, and apply power to the board.

2. On the Tools menu in the Quartus II software, click **Programmer**. The Quartus II Programmer appears and automatically displays the appropriate configuration file (**hello_world.sof)**

3. Click **Hardware Setup** in the upper left corner of the Quartus II Programmer to verify your download cable settings. The **Hardware Setup** dialog box appears.

4. Select the appropriate download cable in the **Currently selected hardware** list. If the appropriate download cable does not appear in the list, you must first install drivers for the cable.

5. Click **Close**.

6. In the **hello_world.sof** row, turn on **Program/Configure**.

7. Click **Start**. The **Progress** meter sweeps to 100% as the Quartus II software configures the FPGA.

At this point, the Nios II system is configured and running in the FPGA, but it does not yet have a program in memory to execute.

Develop Software Using the Nios II SBT for Eclipse

In this section, you start the Nios II SBT for Eclipse and compile a simple C language program. This section presents only the most basic software development steps to demonstrate software running on the hardware system you created in previous sections.

Create a New Nios II Application and BSP from Template

In this section you create new Nios II C/C++ application and BSP projects. Perform the following steps:

1. Start the Nios II SBT for Eclipse. On Windows computers, click **Start**, point to **Programs, Altera, Nios II EDS** *<version>*, and then click **Nios II** *<version>* **Software Build Tools for Eclipse**. On Linux computers, run the executable file *<Nios II EDS install path>***/bin/eclipse-nios2**.

2. If the **Workspace Launcher** dialog box appears, click **OK** to accept the default workspace location.

3. On the Window menu, point to **Open Perspective**, and then either click **Nios II**, or click **Other** and then click **Nios II** to ensure you are using the Nios II perspective.

4. On the File menu, point to **New**, and then click **Nios II Application and BSP from Template**. The Nios II Application and BSP from Template wizard appears.

5. Under **Target hardware information**, next to **SOPC Information File name**, browse to locate the *<design files directory>*.

6. Select **first_nios2_system.sopcinfo** and click **Open**. You return to the Nios II Application and BSP from Template wizard showing current information for the **SOPC Information File name** and **CPU name** fields.

7. In the **Project name** box, type hello_world.

8. In the **Templates** list, select **Hello World**.

9. Click **Finish**.

The Nios II SBT for Eclipse creates and displays the following new projects in the Project Explorer view, typically on the left side of the window:

■ **hello_world**—Your C/C++ application project

■ **hello_world_bsp**—A board support package that encapsulates the details of the Nios II system hardware

Compile the Project

In this section you compile the project to produce an executable software image. For the tutorial design example, you must first adjust the project settings to minimize the memory footprint of the software, because your Nios II hardware system contains only 20 KB of memory.

Perform the following steps:

1. In the Project Explorer view, right-click **hello_world_bsp** and click **Properties**.

The **Properties for hello_world_bsp** dialog box appears.

2. Click the **Nios II BSP Properties** page. The **Nios II BSP Properties** page contains

basic software build settings (it may take a while).

3. Adjust the following settings to reduce the size of the compiled executable:

a. Turn on **Reduced device drivers**.

b. Turn off **Support C++**.

c. Turn off **GPROF support**.

d. Turn on **Small C library**.

e. Turn off **ModelSim only, no hardware support**.

4. Click **OK**. The BSP regenerates, the **Properties** dialog box closes, and you return to the Nios II SBT for Eclipse.

5. In the Project Explorer view, right-click the **hello_world** project and click **Build Project**.

The **Build Project** dialog box appears, and the Nios II SBT for Eclipse begins compiling the project. When compilation completes, a "hello_world build complete" message appears in the Console view.

The console provides info on the size of your program (hello_world.elf), which is the total size of the code and the initialised data, plus info on how much memory has been left free for stack and heap.

Run the Program on Target Hardware

In this section you download the program to target hardware and run it. To download the software executable to the target board, perform the following steps:

1. Right-click the **hello_world** project, point to **Run As**, and then click **Nios II Hardware**. The Nios II SBT for Eclipse downloads the program to the FPGA on the target board and the program starts running.

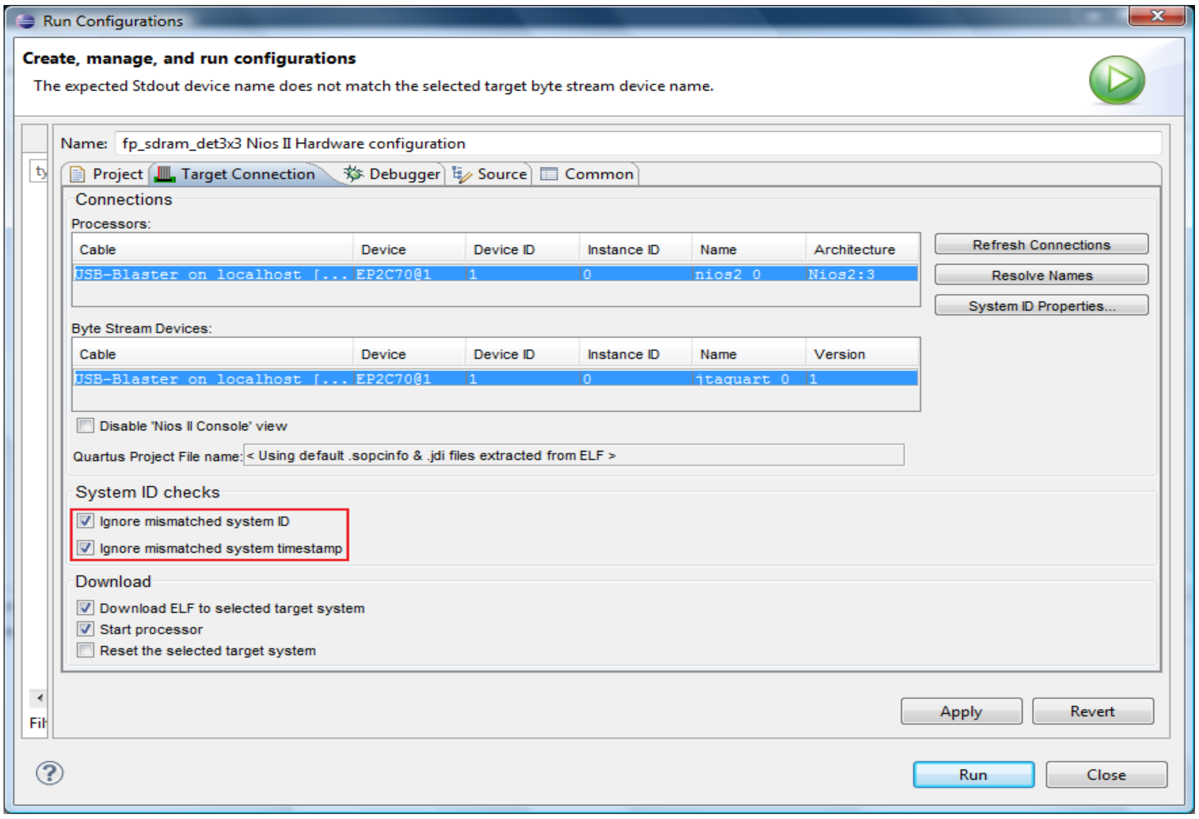
If the **Run Configurations** dialog box appears, verify that **Project name** and **ELF file name** contain relevant data, then click **Run**.

If, and only if, you encounter problems launching your application on the FPGA:

[Target Connection]: Connected system ID hash not found on target at expected base address.

,you may tick the boxes to ignore the system's timestamp and/or system's ID, as shown below.

You are advised to run both *binary_counter* and *hello_world* examples to familiarise yourselves with the system.



Run Configurations

Create, manage, and run configurations

The expected Stdout device name does not match the selected target byte stream device name.

Name: fp_sdram_det3x3 Nios II Hardware configuration

Project | Target Connection | Debugger | Source | Common

Connections

Processors:

Cable	Device	Device ID	Instance ID	Name	Architecture
USB-Blaster on localhost [...	EP2C70@1	1	0	nios2_0	Nios2:3

Byte Stream Devices:

Cable	Device	Device ID	Instance ID	Name	Version
USB-Blaster on localhost [...	EP2C70@1	1	0	jtaguart_0	1

☐ Disable 'Nios II Console' view

Quartus Project File name: < Using default .sopcinfo & .jdi files extracted from ELF >

System ID checks

☑ Ignore mismatched system ID

☑ Ignore mismatched system timestamp

Download

☑ Download ELF to selected target system

☑ Start processor

☐ Reset the selected target system

If you get the message "Downloading ELF Process failed", then this may be for a number of reasons. Unfortunately, the tools (even though they are commercial tools, have bugs). Possible steps to solve the problem:

1. Make sure that the Programmer window from Quartus is closed.

2. You Quartus design may be wrong. Check the connections carefully and also check whether you have connected the reset input to Vcc.

3. Restart the board and Eclipse. Try to resynthesize the design and downloaded again to the board.

When the target hardware starts running the program, the Nios II Console view displays character I/O output.

2. Click the **Terminate** icon (the red square) on the toolbar of the Nios II Console view to terminate the run session. When you click the **Terminate** icon, the Nios II SBT for Eclipse disconnects from the target hardware.

You can edit the **hello_world.c** program in the Nios II SBT for Eclipse text editor and repeat these two steps to witness your changes executing on the target board. If you rerun the program, buffered characters from the previous run session might display in the Console view before the program begins executing.

The final generated system has an Arithmetic Logic Unit (ALU) that supports fixed-point arithmetic, where all the arithmetic operators are implemented using LUTs. The FPGA on the DE0 board is the Cyclone III EP3C16F484C6 FPGA. Make sure you keep track of your NIOS settings throughout the coursework.

In summary, you have designed a system in Quartus, and then moved on to program the system through Eclipse. If you make any changes in the hardware, you need to regenerate the BSP file. Otherwise, you can create new SW programs targeting the same HW without going back to Quartus.

🛠 Record the FPGA resources used by your baseline system (i.e. Total logic elements, Total memory bits, Embedded Multiplier 9-bit elements, and Total PLLs).

Task 2: Computing a simple function

Now that you have your system running on the FPGA, you can start using it to evaluate mathematical expressions.

🛠 The first task is to write a function that takes as input a vector *x* and its length *N*, and returns *y* where $y = \sum_{i=1}^{N} x_i + x_i^2$. The elements of the input vector are single-precision (floats) and are in the range between 0 and 255.

Starting from the **Hello World** template, modify the hello_world.c file as follows:

```
#include <stdlib.h>
#include <sys/alt_stdio.h>
#include <sys/alt_alarm.h>
#include <sys/times.h>
#include <alt_types.h>
#include <system.h>
#include <stdio.h>
#include <unistd.h>

// Test case 1
//#define step 5
//#define N 52

// Test case 2
#define step 0.1
#define N 2551

//Test case 3
//#define step 0.001
//#define N 255001


// Generates the vector x and stores it in the memory
void generateVector(float x[N])
{
         int i;
         x[0] = 0;
         for (i=1; i<N; i++)
                  x[i] = x[i-1] + step;
}

float sumVector(float x[], int M)
{
         // YOUR CODE GOES HERE
}

int main()
{
  printf("Task 2!\n");

  // Define input vector
  float x[N];

  // Returned result
  float y;

  generateVector(x);

  // The following is used for timing
  char buf[50];
  clock_t exec_t1, exec_t2;

  exec_t1 = times(NULL); // get system time before starting the process

// The code that you want to time goes here
  y = sumVector(x, N);

  // till here
  exec_t2 = times(NULL); // get system time after finishing the process

  gcvt((exec_t2 - exec_t1), 10, buf);
  alt_putstr(" proc time = ");        alt_putstr(buf);   alt_putstr(" ticks  \n");
  // printf could be used if there was enough memory

  int i;
  for (i=0; i<10; i++)
          y = y/2.0;
  gcvt(((int) y), 10, buf);
  alt_putstr(" Result (divided by 1014) = "); alt_putstr(buf);
  // printf could be used if there was enough memory



  return 0;
}
```

Complete the code in sumVector() function and compute the results for the three test cases presented in the beginning of this coursework. Increase the onchip memory as much as you can to accommodate the maximum problem size.

<u>**NOTE: You should assume that the vector *x* is stored in the memory prior calling your function. Not all three test cases need to be in the memory of the system at the same time. This assumption applies to the whole project.**</u>

Provide answers to the following questions:

How many resources does your system occupy on the FPGA?

Is your system able to evaluate the given function for all test cases? Why not?

What is the size of your NIOS II application for the different vector sizes and what is the maximum memory required?

Compare your results to Matlab. Do the results agree? If not, why not?

What is the time required to evaluate the sumVector() function for each test case? If you require more accuracy in the calculation of the time required by sumVector() function, you can call the function a number of times and get the average time spend in the function.

In designing HW systems, there is a relationship between the HW resources used by your system and the performance that you get out of it. If we assume that the accuracy is fixed and is not affected by the HW resource allocation, then the performance in your case is the latency of the function sumVector(). The best design is the one that provides the least latency for the least possible resources. Where is your current design in that space?

More detailed information about programming NIOS II in EDS is offered here:

📖 http://www.altera.com/literature/hb/nios2/n2sw_nii52017.pdf

To debug your application running on NIOS II use 🐞 instead of ▶. Make use of the perspectives to switch between NIOS II project and debug (**Window → Open perspective**). You're encouraged to make use of the available debug tools, e.g. watch and breakpoint, to facilitate the development of the software.

☺ Good to know:

The printf() command in NIOS by default does not support floating point printing on the screen in order to reduce the size of the generated code. As you have only available the on-chip memory to store your program, the result can be produced by printing to the output a scaled down integer. Thus, if result is your floating point result, your can produce the result as follows:

```
For (int i=0; i<<10; i++)

        resultf = resultf/2.0;

printf("Result: %d \n",(int) result);
```

☺ Good to know:

Since we are interested in accelerating a certain computational process we need to be able to know how long it takes to execute that process. The time required to complete any statement/block of statements can be determined using the following statements in C code:

```
char buf[40];

clock_t exec_t1, exec_t2;

exec_t1 = times(NULL); // get system time before starting the process
*** your C statement(s) here ***
exec_t2 = times(NULL); // get system time after finishing the process

gcvt((exec_t2 - exec_t1), 10, buf);
alt_putstr(" proc time = ");     alt_putstr(buf);    alt_putstr(" ticks  \n");
```

which requires the following includes:

```
#include <stdlib.h>
#include <sys/alt_stdio.h>
#include <sys/alt_alarm.h>
#include <sys/times.h>
#include <alt_types.h>
#include <system.h>
#include <stdio.h>
#include <unistd.h>
```

It also requires the Interval Time block to be in your system and running in the Hardware Abstraction Layer (HAL). Please refer to the tutorial for more details about this. Please note that the above code may require extra memory that your system may not have.

☺ Good to know:

See command `nios2-elf-size` to get the size of your application. Alternatively, you can look at the .map file, or at the output of the building project process.

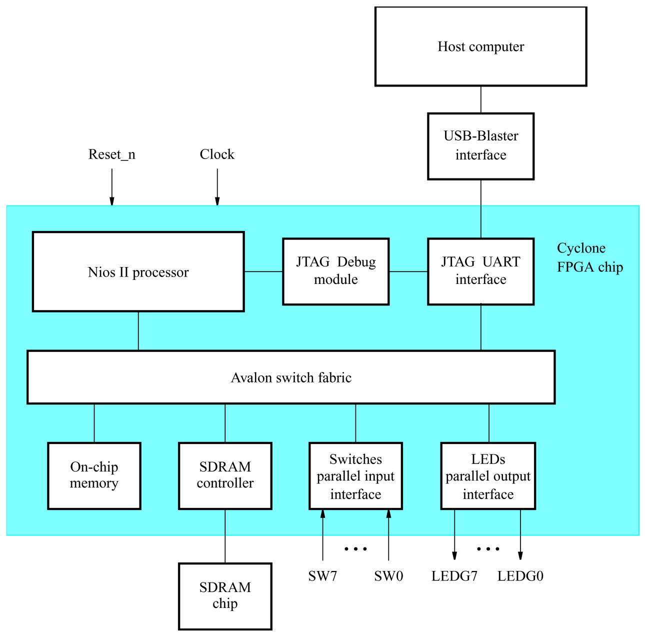
Read the reference about application profiling at the end of this hand-out.

Task 3: Storing the Program and Data on External Memories

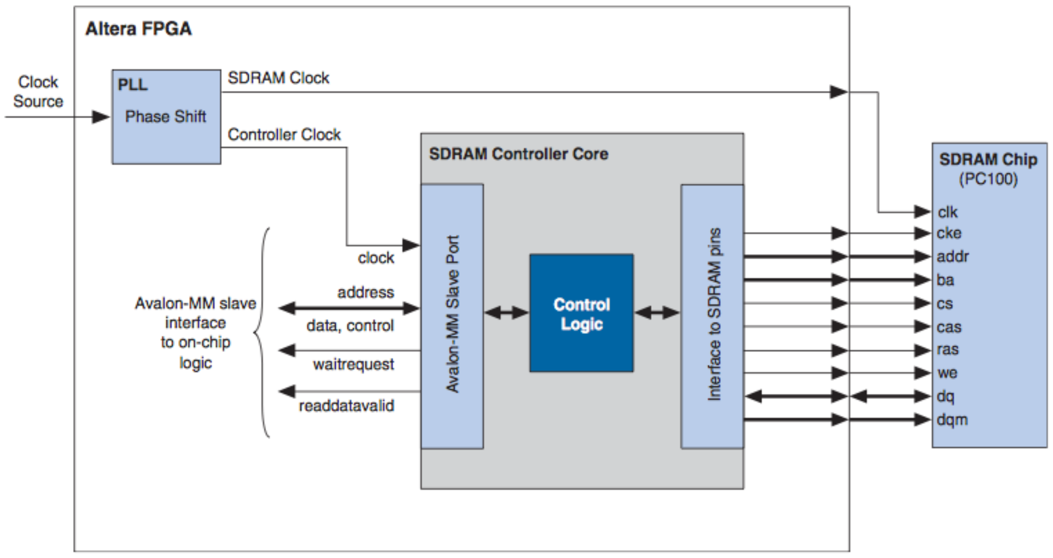
The system that you created has its application stored on on-chip memory. Despite the fact being the fastest one available on an FPGA, it is the smallest and the most expensive one available in your system. On the DE0 boards there are different types of memories, including SDRAM memory. The main benefit of using SDRAM is that it has much more capacity and it has lower cost. That's why it is often used in real-life applications. The main drawback of using these memories is their interface, which requires a controller, and a consequent penalty on the system's complexity and performance, when compared to on-chip memory.

In this task, we will assess how the SDRAM memory allows to address problems with large memory requirements but also how its performance impacts the overall performance of the system.

The Altera DE0 board contains an SDRAM chip that can store 8 Mbytes of data. This memory is organized as 1M x 16 bits x 4 banks. The SDRAM chip requires careful timing control. To provide access to the SDRAM chip, Qsys implements an SDRAM Controller circuit. This circuit generates the signals needed to deal with the SDRAM chip. The figure below shows the resulting system by including the SDRAM controller (Please note that the "Switches parallel input interface" block is missing from your current system). You can notice that all the connections to on-chip memory and SDRAM controller are through the Avalon switch fabric.



The signals needed to communicate with this chip are shown in the figure below. All of the signals, except the clock, can be provided by the SDRAM Controller that can be generated by using Qsys. The clock signal is provided separately. It has to meet the clock-skew requirements (it will be explained in the lectures). Note that some signals are active low, which is denoted by the suffix N. Due to clock skewness a PLL is needed to generate a shifted version of the clock that drives the SDRAM.



In order to include the SDRAM controller in your existing system, open your current design in Qsys and select Memories and Memory Controllers > External Memory Interfaces > SDRAM Interfaces > SDRAM Controller and click Add. In the new window

Set the Data Width parameter to 16 bits and leave the default values for the rest. Since we will not simulate the system in this tutorial, do not select the option Include a functional memory model in the system testbench. The rest of the parameters of the SDRAM controller for the DE0 are given below.

```
Memory Profile:
bits: 16; chip select: 1; banks: 4; row: 12; col:  8;
Timing:
cas:  3;  init:  2;  refresh:  15.625;  delay:  100;  t_rfc:  70;  t_rp:  20;  t_rcd:  20;
t_ac: 5.5; t_wr: 14.0;
```

Click Finish. Now, in your final system, there will be an sdram module added to the design. Rename the module to **sdram,** and make the clk and reset connection as in the other modules. Connect the s1 signal to both cpu.data_master and cpu.instruction_master. Export the *wire* signal by double clicking on the export column.

So far, we have instantiated an SDRAM controller in the design, and have made the necessary connections to the rest of the design. We have also told to the tool to export the wire signal, which will be connected to the SDRAM memory directly.

The next stage is to integrate this memory in the memory address space of the system and inform the CPU where our reset_vector and exception_vector addresses are.

Select the command System > Assign Base Addresses. Observe that the SOPC Builder assigned the base address 0x00800000 to the SDRAM. To make use of the SDRAM, we need to configure the reset vector and exception vector of the Nios II processor. Right-click on the cpu_0 and then select Edit. Select sdram_0 to be the memory device for both reset vector and exception vector. Click Finish to return to the main window regenerate the system.

Close Qsys and go back to Quartus. Right click on your system and select Update System or Block. This should now reflect the extra wire connections from the SDRAM controller that we selected to export. Connect these to the relevant DRAM output pins already existing in your design.

Using the Clock Signals IP Core

The clock skew depends on physical characteristics of the DE0 board. For proper operation of the SDRAM chip, it is necessary that its clock signal, DRAM_CLK, leads the Nios II system clock, CLK_50, by 3 nanoseconds.

This can be accomplished by using a phase-locked loop (PLL) circuit which can be manually created using the MegaWizard plug-in.

To add the PLL IP core, right click somewhere in your design, and select Insert > Symbol. Select the **altpll** core and the MegaWizard window will appear. Click Next.

There are many parameters that you can set in the block, but what you need is to provide an input clock of 50MHz and provide two output clocks at the same frequency but one of them with some phase difference enough to compensate for the clock skew.

Provide the 50MHz clock input, without changing any of the other settings and select next. Then click on the Output Clocks tab. C0 clock should be kept the same as the input clock (i.e. freq and zere phase diff). Click next to go tp c1 settings and select to use that clock, having a 50MHz freq, but the requested phase diff to be -2.55ns (You may have to fine-tune the PLL time shift to a different value for your particular DE0 board, but the value should be between -2.62ns amd -2.50ns (a phase shift of about -46 degrees for 50MHz clock). No other clock is needed, so click on Finish. (There is no need to have areset and lock signal in your PLL). One indicator of such problem is the absence of output in the console terminal, when you run the application or the fact that ELF failed to download/verify.

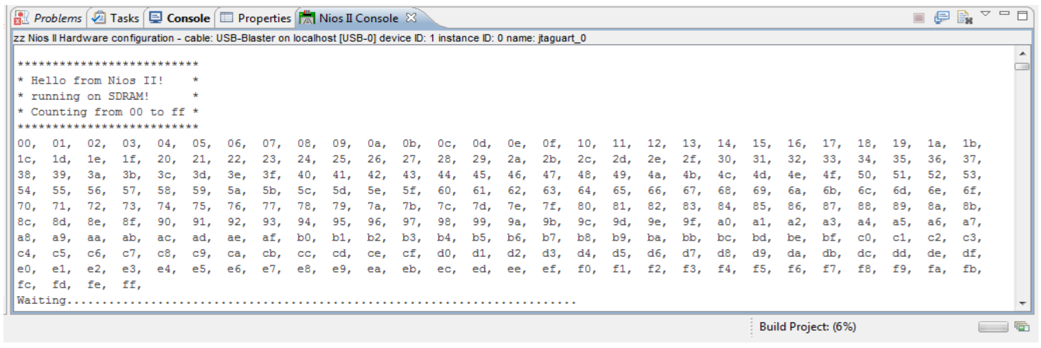
Place the new block in the design and make the necessary connections to the clocks. Which clock output from the PLL should drive which module?

You also can place the PLL inside Qsys. The only difference is if you have to change the parameters of the PLL, you will have to re-generate your Qsys system, whereas if you have the PLL in your top design, you'll only need to re-synthesize the design.

Please note that SDRAM is very sensitive to clock variations. It is recommended that you use an external PLL in your design.

You can read more about this here: http://www.altera.com/literature/ug/ug_embedded_ip.pdf

To assess the correct functioning of your system with SDRAM, change your application so that it prints a different statement in the console output. Below there's the example of the **count_binary** template application. Please note that you can use the whole support for the C++ language as you have enough memory to store your programs (i.e. printf to print the floating point number).



```
zz Nios II Hardware configuration - cable: USB-Blaster on localhost [USB-0] device ID: 1 instance ID: 0 name: jtaguart_0

**************************
* Hello from Nios II!    *
* running on SDRAM!      *
* Counting from 00 to ff *
**************************
00,  01,  02,  03,  04,  05,  06,  07,  08,  09,  0a,  0b,  0c,  0d,  0e,  0f,  10,  11,  12,  13,  14,  15,  16,  17,  18,  19,  1a,  1b,
1c,  1d,  1e,  1f,  20,  21,  22,  23,  24,  25,  26,  27,  28,  29,  2a,  2b,  2c,  2d,  2e,  2f,  30,  31,  32,  33,  34,  35,  36,  37,
38,  39,  3a,  3b,  3c,  3d,  3e,  3f,  40,  41,  42,  43,  44,  45,  46,  47,  48,  49,  4a,  4b,  4c,  4d,  4e,  4f,  50,  51,  52,  53,
54,  55,  56,  57,  58,  59,  5a,  5b,  5c,  5d,  5e,  5f,  60,  61,  62,  63,  64,  65,  66,  67,  68,  69,  6a,  6b,  6c,  6d,  6e,  6f,
70,  71,  72,  73,  74,  75,  76,  77,  78,  79,  7a,  7b,  7c,  7d,  7e,  7f,  80,  81,  82,  83,  84,  85,  86,  87,  88,  89,  8a,  8b,
8c,  8d,  8e,  8f,  90,  91,  92,  93,  94,  95,  96,  97,  98,  99,  9a,  9b,  9c,  9d,  9e,  9f,  a0,  a1,  a2,  a3,  a4,  a5,  a6,  a7,
a8,  a9,  aa,  ab,  ac,  ad,  ae,  af,  b0,  b1,  b2,  b3,  b4,  b5,  b6,  b7,  b8,  b9,  ba,  bb,  bc,  bd,  be,  bf,  c0,  c1,  c2,  c3,
c4,  c5,  c6,  c7,  c8,  c9,  ca,  cb,  cc,  cd,  ce,  cf,  d0,  d1,  d2,  d3,  d4,  d5,  d6,  d7,  d8,  d9,  da,  db,  dc,  dd,  de,  df,
e0,  e1,  e2,  e3,  e4,  e5,  e6,  e7,  e8,  e9,  ea,  eb,  ec,  ed,  ee,  ef,  f0,  f1,  f2,  f3,  f4,  f5,  f6,  f7,  f8,  f9,  fa,  fb,
fc,  fd,  fe,  ff,
Waiting.....................................................................
```

Evaluate the performance of your system using the same test cases as in the introduction of this hand-out and discuss the obtained results. Answer the following questions:

How long does it take to complete the evaluation of the function for each of the test cases?

What is the size of your NIOS II application for the different vector sizes?

How many resources does the final system occupy on the FPGA?

What is the impact of increasing the cashe memory size of the NIOS processor to the required resources and the achieved latency of the system?

Hints:

- If you experience problems running your application on the updated system, try to create a new Board Support Package (BSP) project and copy only the C source files from your previous project**. It is advised to create a new project with BSP**. Avoid using existent BSPs after updating your system.
- If you copy the folder of your previous project, pay attention to the paths on files: **create-this-bsp** and **settings.bsp**. They may not be relative to the new project folder!
- Stop the execution of your application before trying to re-program the FPGA, as you may lose connection to it.

Task 4: Evaluate a More Complex Mathematical Expression

By completing Task 3, you should have a good understanding on how an embedded digital system is constructed and how certain design decisions affect its performance. Your basic embedded system is based on the NIOS processor and has a large amount of memory (8MB). From now on, we will focus on the initial objective of the project, which was to evaluate the following function and design accelerators to improve the design's performance.

$$f(x) = \Sigma(0.5*x + x^2*\cos(\mathrm{floor}(x/4)-32))$$

As now your system includes access to enough memory, the math.h library can be utilised to help with the evaluation of the cosine function.

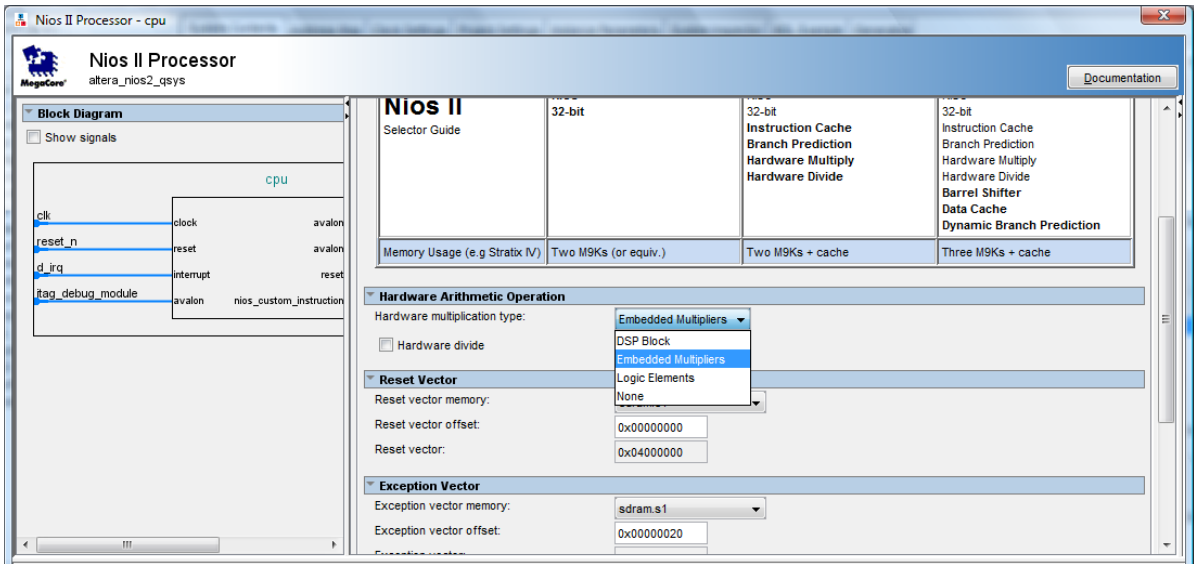
Make use of the math.h library and implement the new mathematical expression in software.

Run the application for all three test cases and report the performance of your design regarding: HW resources, latency to evaluate the function in seconds, and accuracy of the final result (compared to a MATLAB double precision implementation).

Task 5: Add Multiplier Support

When you first created your NIOS II it had no multiplication support, which means that the floating-point multiplications have been emulated in software at the expense of fixed-point adders.

Now you're going to add hardware support for multiplication. NIOS II provides two possibilities: Logic Elements or Embedded Multipliers (same as DSP block).



Nios II Selector Guide	32-bit	32-bit **Instruction Cache** **Branch Prediction** **Hardware Multiply** **Hardware Divide**	32-bit Instruction Cache Branch Prediction Hardware Multiply Hardware Divide **Barrel Shifter** **Data Cache** **Dynamic Branch Prediction**
Memory Usage (e.g Stratix IV)	Two M9Ks (or equiv.)	Two M9Ks + cache	Three M9Ks + cache

Hardware Arithmetic Operation

Hardware multiplication type: Embedded Multipliers (DSP Block / Embedded Multipliers / Logic Elements / None)

Hardware divide [ ]

Reset Vector

Reset vector memory:

Reset vector offset: 0x00000000

Reset vector: 0x04000000

Exception Vector

Exception vector memory: sdram.s1

Exception vector offset: 0x00000020

LUT-based multipliers are often used on FPGAs due to the flexibility of their word-length configuration, allowing faster multiplications at the expense of computing fewer bits than generic fixed word-length hard-multipliers. In the case of floating-point arithmetic the significand (also known as mantissa) has 23 bits, which is long enough to benefit from the performance of dedicated embedded multipliers.

So far you have been using floating-point arithmetic with non-floating-point units on your system. The software has been emulating these functions relying on a basic ALU implemented with generic LUTs (FPGA's basic logic elements). The FPGA that you are using, Cyclone III http://www.altera.com/literature/lit-cyc3.jsp has dedicated hardware multiplier blocks that can be used to do fixed-point multiplications faster. Please note that you just add support for integer multiplication and the floating-point arithmetic has to be emulated again (signs, exponents, rounding and packing).

🛠 In order to improve the performance of your system, we are going to add HW support for fixed-point multiplications.

Go back to the Qsys, and edit the **cpu**. Going through all the possibilities from the drop-down menu in Qsys, update the system and compare the new performance results with the previous version of your system for all three test cases. Report the execution time (i.e. latency) of the function that computes F(x), resource utilisation, code size, and error between the result of your system and the reference double precision Matlab implementation for each option in the drop-list and for each test case.

Assuming that maximum performance is required by the application, which design choice should be selected? Justify your choice.

Task 6: Add Hardware Floating-Point Units

FLOATING-POINT (BACKGOURND)

In many applications there is a requirement to process data with large dynamic range and approximate data that takes values from the real number set. A possible way to represent the above numbers in a physical system is to use the "universally adopted" IEEE-754 floating-point standard.

The intention of the IEEE Standard for Binary Floating-Point Arithmetic, also known as IEEE-754, is to specify floating-point formats, arithmetic, conversions and exceptions. Floating-point numbers have three components: sign, significand and exponent, generically given by the following equation:

$$x = \pm \text{significand} \times 2^{\text{exponent}}$$

The sign is represented with one bit indicating whether the number is positive (0) or negative (1). The significand represents the fractional part of the number, in a fixed-point format. The exponent indicates the magnitude of the number, and has an embedded biased value. The number of bits of the significand determines its precision. In the IEEE standard, the significand is a real number in the interval [1; 2[. The integer component, value 1 is not included, so the significand contains only the fractional part of the floating-point value. The standard defines two formats: single precision (32-bit) and double precision (64-bit). In this coursework only the single precision format is considered. This format uses 8 bits to represent the exponent and 23 bits + 1 hidden bit for the significand, as shown in the figure below.

sign	exponent	significand
1 bit	**8 bits**	**23 bits**

Some values encoded using single precision floating-point representation:

Exponent	Significand	Value
0	0	±0
0	Not 0	$\pm 2^{-126}$ x(0.significand) [denomal]
1-254	-	$\pm 2^{(\text{exponent}-127)}$ x(1.significand)
255	0	±∞
255	Not 0	Not a Number

Example: π = 3.14159265358979323846264338327950

To be between 1 and 2 we need to divide it by two, which means the exponent needs to be 127 + 1.

With 23 bits left we represent digits from the left to the right of 0.57079632679489661923132169163975

sign	exponent	significand
0	10000000	10010010000111111011011

So far there's no support for floating-point arithmetic in NIOS II. All your floating point operations are emulated through SW. Now you are going to add floating-point adder, subtraction and multiplier hardware, so that the elementary arithmetic operations used by the function can be performed faster, thereby reducing the execution time, in the expense of dedicating more FPGA resources.

In order to implement such functionality in dedicated hardware you'll need to design a hardware block that performs this functionality, attach it to a bus in the system so the processor can access it, and then create a custom instruction so you can access it from your software.

To design the hardware block in Quartus II you can use schematic capture[1], Verilog or VHDL. Before you attach your HW block to the NIOS system, you are advised to make a functional simulation of your hardware block to verify that it works correctly. You can do simulations using Modelsim[2] or Active-HDL. After that, you will add it to the system using Qsys (**Component Library panel→Project→New Component**).

☺ Good to know:

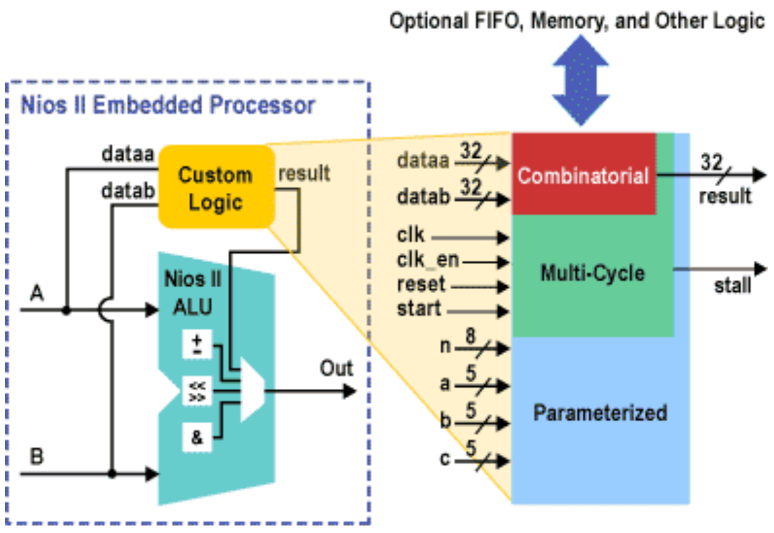
In Qsys, you can add **Floating-Point Hardware** from MegaWizard to your system, and then connect it to the **Custom Instruction Master** port on you processor. In this case the floating-point hardware will automatically be mapped to a custom instruction and all floating-point arithmetic in software will be using this hardware block. The main limitation of using this is that you don't control latency or any other parameters or functionality made available through the MegaWizard interface. Also, you won't be able to reuse these units inside your hardware blocks bypassing the CPU. Also, the focus is to learn how to design a new custom instruction in order to be able to design future acceleration units. **As such, the Floating-Point Hardware in Qsys is not permitted as a design choice in this project.**

[1] Recommended (w/ help available), but you can use any alternative.

[2] Again, recommended (w/ help available), but you can use another. There's a Modelsim tutorial at the end of this hand-out.

NIOS II CUSTOM INSTRUCTIONS

NIOS II supports different types of custom instructions depending on the type of hardware block connected to it. The following diagram shows how the custom logic is interfaces with the rest of the system.



The supported custom instructions/logic blocks are:

- Combinatorial: single clock cycle logic block; In this case, your HW block should be able to produce a valid results within one clock cycle.
- Multi-cycle: fixed or variable number of clock cycles to complete execution;
- Extended: block capable of performing multiple operations (e.g. ALU); In this case, your block has an extra input that specifies the operation to be performed.
- Internal register file: logic blocks that access internal register files for I/O (e.g. Multiply-Accumulate unit);
- External interface: logic blocks that interface to logic outside of the Nios II processor's data-path.

A high level view of the steps that you need to follow in order to create your own custom instruction component is shown below:

- Do a block diagram / Verilog / VHDL with your circuit;
- If you're doing block diagram, generate Verilog from it;
- Simulate your block using Modelsim or one of the other tools. If it performs as intended move on, otherwise revise your circuit design;
- Add the new block to your system as a new component;
- Synthesise your system;
- Get FPGA resources, latency and throughput;
- Add the "software interface" for your new block.

Add custom instruction tutorial

In order to familiarise yourself with the process of adding a custom instruction to your existing system, the following section provides a tutorial on how to map a simple one-cycle fixed point multiplier to your system as a custom instruction. This tutorial should be followed in parallel with the ug_nios2_cusotm_instruction.pdf document (http://www.altera.com/literature/ug/ug_nios2_custom_instruction.pdf) where more detailed information on the process is provided.

Firstly we need to identify the type of the custom instruction. As we need to implement a single one-cycle multiplier, the *Combinatorial* type is appropriate.

Type	**Application**	**Hardware ports**
Combinational	Single Clock cycle custom logic blocks	dataa[31:0] datab[31:0] result[31:0]

Details on the available types can be found in Table 1-1 of http://www.altera.com/literature/ug/ug_nios2_custom_instruction.pdf

Design the module

Create a new project in Quartus, and insert a new symbol from the Megafunction library. You should select the LPM_MULT. Configure your multiplier to take two 32 bit inputs. Select your multiplier to be unsigned and without any pipeline stages (no clock). Your final multiplier should produce a 32 bit output, which are the 32 LSBs from the LPM_MULT component.

Verify its functionality

After having instantiating your IP code, add input and output ports and simulate it to verify its functionality. The simulation can be done though the Quartus environment, or you can use ModelSim (there is a tutorial on this at the end of this document). Please note that you have kept only part of the result (32 LSBs).

Add the module to the existing system

In order to add your custom HW to the rest of the system, you need first to create an HDL design file from your top-level schematic file. Next, open your system in Qsys and click on *New Component* in the Library window. Provide the name of your component, and in the Files tab add the vhdl or Verilog files that you have created for your component (i.e. these are the top-level vhdl or Verilog file and the vhdl or Verilog file of the lpm_mult). Click Top Level Module and select the name of the top-level module of your custom instruction logic.

To configure the custom instruction signal type, follow these steps:

1. Click on the Signals tab.

2. For each port listed in the tab, follow these steps:

a. Select the port.

b. In the Interface column, select the name of the interface to which you want to assign the port. For the design example, for the first port you configure, select New Custom Instruction Slave. For the remaining ports, select the name created for the first port, which is nios_custom_instruction_slave. These selections ensure that the ports appear together on a single interface.

c. In the Signal Type column, select the appropriate signal type for the port.

To set up the custom instruction interfaces, follow these steps:

1. Click on the Interfaces tab.
2. Click on **Remove Interfaces With No Signals** button (if it is active).
3. Ensure that a single interface remains, with Name set to the name in the Signals tab. For the design example, maintain the interface name nios_custom_instruction_slave.
4. Ensure the Type for this interface is Custom Instruction Slave.
5. For Clock Cycles, you need to type 0 for a variable multicycle type custom instruction, and otherwise type the number of clock cycles your custom instruction logic requires. In our case we need to type 0, but this has already been picked up by the tool.
6. In the general case, for Clock Cycle Type, type Variable for a variable multicycle type custom instruction, Multicycle for a fixed multicycle type custom instruction, or Combinatorial for a combinational type custom instruction.
7. For Operands, type the number of operands for your custom instruction.

If your custom instruction logic requires additional interfaces, either to the Avalon-MM fabric or outside the Qsys system, you can specify the additional interfaces in the Interfaces tab. Our example does not require additional interfaces.

Click Finish and Save. In the Qsys Component Library, under Library, select you custom module and click add to add the new instruction to the Qsys system. In the Connections panel, connect the new component component's nios_custom_instruction_slave interface to the cpu component's custom_instruction_master interface.

Generate the new NIOS system, return to Quartus, add the custom instruction's directory through the **Projects -> Add/Remove Files in Project**, and then recompile the whole design in Quartus.

Check whether the new design meets timing.

Accessing the Custom Instruction from Software

Adding a custom instruction to a Nios II processor results in a significant change to the Qsys system. In this section, you create and build a new software project using the Nios II software build flow, and run the software that accesses the custom instruction.

Create a new software project using the Hello World template for your system and name it *hello_world_custom_instr*. After the file generation has been finished, click on *hello_world_custom_instr_bsp* and open the *system.h* file. Verify that a custom instruction macro has been generated as follows:

```
#define ALT_CI_CUSTOM_MUL_0(A,B) __builtin_custom_inii(ALT_CI_CUSTOM_MUL_0_N,(A),(B))

#define ALT_CI_CUSTOM_MUL_0_N 0x0
```

The second line provides the opcode for your custom instruction, and you can verify that this is the opcode assigned to your instruction by Qsys. The first line describes the custom instruction (input arguments and return values) and the associated custom instruction built-in function (Appendix B of ug_nios2_custom_instruction.pdf).

Without making any changed to the template, build and run the application on the actual hardware to verify its functionality.

Change the code of hello_world.c file as follows and verify its functionality.

```
#include <system.h>
#include <stdio.h>

int main()
{
  printf("Hello from Nios II!\n");

  int a, b, c, d;
  a = 2;
  b = 4;
  c = a*b;
  printf("Multiplication result: %d\n", c);

  d = ALT_CI_CUSTOM_MUL_0(a,b);
  printf("Multiplication result from custom instr: %d", d);

  return 0;
}
```

Please note that you need to include *the system.h* file in order to access the custom instruction.

These concludes the tutorial on how you can instantiate a new hardware block in your system and create a new custom instruction to allow the NIOS processor to access it. More detailed information can be found in:

📖 http://www.altera.com/literature/ug/ug_nios2_custom_instruction.pdf

(Chapters 1, 2 and 4).

📖 You can find more information on Altera's support on floating point IP cores and the necessary conversions on the following document: http://www.altera.com/literature/ug/ug_altfp_mfug.pdf

Add custom instruction to your system

Having gone through the previous section, you should be familiar with the process of adding custom instructions to a NIOS system. The evaluation of the target function involves heavily use of floating-point addition, subtraction and multiplication, which at the moment are emulated through SW. As the target is to accelerate the evaluation of the function, the aim of this section is to map this functionality to HW.

🛠 Implement the custom logic blocks for floating-point adder, subtracter and multiplier operations in order to support hardware acceleration of the above operations. Design the HW block, verify it, and then integrate it to the rest of the system. Make sure that you select a suitable

Which type of custom instruction did you choose (i.e. Combinatorial, Multi-cycle,…)? Justify your choice.

What is the FPGA resources allocated for the new block? What is the throughput and latency of the new HW block? How the latency of your HW block impacts the overall performance of the system?

Compare the new performance results with the best performance achieved from the previous version of your system for all three test cases. Report the execution time (i.e. latency) of the function that computes F(x), resource utilisation, code size, and error between the result of your system and the reference double precision Matlab implementation for each test case.

In the case where you have pipelined your design, please comment on how well the pipeline is utilised? Justify your answers.

ⓘ VERIFICATION: GENERATING A SYSTEM, SYNTHESISING AND TESTING IT (A.K.A. TRAIL-AND-ERROR) IS TIME CONSUMING. YOU SHOULD VERIFY YOUR DESIGN THROUGH SIMULATION BEFORE TRYING RUNNING IT ON THE FPGA. THIS IS HOW REAL-LIFE SYSTEMS ARE DESIGNED, AND YOU SHOULD FOLLOW THIS PRACTICE THROUGHOUT THE COURSEWORK.

Hints:

- Organize your hardware blocks in a subfolder named "./IP" in your project folder;
- Create a new block diagram file and place your floating-point units there along with the necessary control logic. Remember to use labels in your connections, this will help to identify them during simulation;
- Pay attention to the interface signals of your block. They'll be different for different types of custom instructions;
- The floating-point arithmetic units are added to your block diagram via **Tools→MegaWizard Plug-In Manager**.
- Use MegaWizard to create your other blocks (LPM_*). Add them to your design, as Qsys will only add the top HDL file to your design. DO NOT add your top block of your design to the project, only the sub-blocks;
- If you're not familiar with either Verilog or VHDL, do your design using block diagram and then convert it to Verilog using **File→Create/Update→Create HDL File From Current File**. While creating a new component in Qsys it will ask for a VHDL or Verilog file. Remember to create a new one, and update your component, every time you update the block diagram. Qsys will create a copy of your HDL file (top of your block) when you create the component to a different location. Updating your HDL file won't update your block in the system. You have to manually copy it and replace it or regenerate the component;
- Keep in mind that Altera's CRC example is different than your floating-point add/sub/mult. You can't *just* follow the steps as they are in the guide; you need to think about what needs to be changed;
- After generating the BSP, verify that you have the macros for your custom instruction in the file **system.h**. **Check if the arguments and return value of __builtin_custom_?n?? agree (see manual appendix b)**. If not, create a new macro in your source file. **DO NOT EDIT system.h!**
- You can create specific header (prototypes of your software functions) and source (code to access your custom instructions) files for your custom instructions, e.g. **custom_instr_fp.h/.c**;
- Replace all floating-point arithmetic operands in your application with calls to your custom application. E.g. `(float)x = (float)a * (float)b;` ➔ `x = cust_fp_mult(a, b);`.
- To interface the hardware block (read/write) in EDS you can use:

  `IOWR(SIMPLE_HW_BASE, register_address_in_your_unit, 32bit_data_value);` and

  `32bit_data_value = IORD(SIMPLE_HW_BASE, register_address_in_your_unit);`

  check Qsys and/or system.h in EDS/BSP for the address of your HW unit.

Debug:

- In case you don't read the expected value from your custom instruction, place a constant at the output of your hardware unit and verify if that value is read by the instruction.
- Whenever you introduce changes in your design, use different constant inputs so you can verify that your design is being updated. Qsys stores a copy of your top HDL file in a different location than your

./ip folder. It is a good idea to have the connections correct in that design before adding it to the Qsys.

- Use the simulation to adjust the correct number of clock cycles required to complete the operation.
- If you fail to have any output from your instruction at all, try to implement a simple one (combinatorial) to verify that you are calling the custom instruction correctly from your software. E.g. load a constant value from it, negate one of the outputs, OR/AND/XOR your inputs, etc.

Task 7: Add Dedicated Hardware Block to compute the inner part of the arithmetic expression

You've seen that you can improve the performance of your system by implementing some of the most computationally demanding operations in hardware. In the present scenario you're going to extend your previous implementation and map the whole arithmetic expression as a custom instruction in your system.

Add the following custom instruction to your system:

$$y = x + x^2 * \cos(\text{floor}(x/4)-32)$$

by mapping all operators to hardware including the implementation of a CORDIC block to implement

`cos(.)`

instead of using the software implementation from `math.h`. Please note that the input to the `cos(.)` is an integer. Do not forget that we are aiming for the best performing design (fastest execution time) but with the smallest resource utilisation.

Please note that a look-up table based implementation is not allowed for the implementation of the

`cos(.)` function, or the use of an existing IP block that performs this function.

Comment on the architecture of your CORDIC HW block. How many stages does it have? How did you select the number of stages? What is the impact on the function evaluation for the 3 test cases?

Describe how data is passed in your new HW block. Comment on its pipeline's utilisation.

Compare the new performance results with the best performance achieved from the previous version of your system for all three test cases. Report the execution time (i.e. latency) of the function that computes F(x), resource utilisation, code size, and error between the result of your system and the reference double precision Matlab implementation for each test case.

You can find more information on Altera's support on floating point IP cores and the necessary conversions on the following document: http://www.altera.com/literature/ug/ug_altfp_mfug.pdf

Task 8: Add Dedicated Hardware Block to compute the arithmetic expression

Throughout this coursework you have been accelerating part of your software by desiging dedicated hardware blocks in the hardware that accelerate part of the required computations. (Hopefully) your application has been computing the targeted arithmetic function faster through all these hardware support.

The aim of this final task is to map the whole function in hardware and achieve as reduce the execution time as much as you can. This is an open-ended task that you are free to use any possible architecture with the only restriction beeing that you have to use the standard version of the NIOS II processor (NIOSII/s).

Some design Hints:

- You need to add a floating-point adder to perform the overall sum in hardware. In this way you can speed up part of the computations.
- Think whether you have taken full use of the pipeline capabilities of your hardware blocks. How often you can push a new value of the x vector to your system? How can you increase its throughput? Do not forget that your NIOS processor blocks after the execution of a custom instruction and waits for the assertion of the done signal to retrive the result and being able to move to the next instruction. State any assumptions that you may have in your implementation.

Comment on the architecture of your final system and on the control and data flow of your design. Describe how and what data is passed in your new HW block. Comment on its pipeline's utilisation.

Compare the new performance results with the best performance achieved from the previous version of your system for all three test cases. Report the execution time (i.e. latency) of the function that computes F(x), resource utilisation, code size, and error between the result of your system and the reference double precision Matlab implementation for each test case.

Debug hint:

- If by any chance your system fails to synthesize with an error similar to this:

```
Error (12006): Node instance "func_accel" instantiates undefined
entity "first_nios2_system_func_accel"
```

  Double click on the error, select the first line of that unit instantiation, and replace:

  `"first_nios2_system_func_accel"` with `"func_accel"`

  This happens because somewhere in the script to generate the NIOS II system, the instantiation of your hardware block got its name concatenated with the name of your system.

MARKING SCHEME

The distribution of marks to the tasks is shown below. Please note that the marking will take into consideration your design approach, your achieved results, as well as the quality of the submitted report.

Task	**Mark**
Tasks: 1 + 2	0 %
Task 3: Storing the Program and Data on External Memories	10 %
Task 4: Evaluate a More Complex Mathematical Expression	10 %
Task 5: Add Multiplier Support	10%
Task 6: Add Hardware Floating-Point units	20%
Task 7: Add dedicated Hardware Block to compute the inner part of the Arithmetic Expression	20%
Task 8: Add dedicated Hardware Block to compute the Arithmetic Expression	20%
Final Presentation + Interview	10%

COURSEWORK SUBMISSION

Proposed work schedule and Intermediate deliverables deadlines

You should register in a logbook all that you have accomplished including results. The logbook doesn't replace the report.

Schedule

Tasks	**Week starting on:**	**Deliverable deadline** **(end of week, Friday@midnight)**
Task 1: NIOS II Setup Task 2: Computing a simple function	18/1	
Task 3: Storing the Program and Data on External Memories Task 4: Evaluate a More Complex Mathematical Expression	25/1	Report 1 (Tasks:1-2)
Task 5: Add Multiplier Support	1/2	
Task 6: Add Hardware Floating-Point Units	8/2	Report 2 (Tasks: 3-5)
	15/2	
Task 7: Add Dedicated Hardware Block to Compute the inner part of the Arithmetic Expression	22/2	
Task 8: Add Dedicated Hardware Block to Compute the	29/2	

Arithmetic Expression		
	7/3	
Examination Week (The oral examination will take place on Wednesday 16th March and/or Friday 18th March).	14/3	Report 3 (Tasks: 6-8)

Only Reports 2 and 3 are assessed. Report 1 should be submitted for feedback. If you are late in submitting your report, the usual College penalty on late submission is applied. In the case of delaying Report 1, a 10% penalty will be applied on the final overall mark.

Reports

Each group needs to submit three reports on their project according to the schedule above.

The reports should contain the following:

a) Answers to any questions raised in the project hand-out, e.g. performance, FPGA resources, etc.;

b) Any related design in block diagram;

c) A detailed justification of the design choices that you have made;

d) Any limitations of your design;

e) An elaboration of what you would have done differently if you were doing the project from the beginning (for example you may have found that you took a wrong design decision but it is too late to go back and fix it. It is important to elaborate on this (why you believe it is a wrong decision and what would be a better one).

Please note that special attention will be given on the justification of your design choices and how you have approached the problem. So please provide enough details around these issues.

Also note that the report should not contain any background info like literature review on NIOS. However, the report should provide information on why you have chosen specific algorithms for the calculation of the arithmetic expression.

Below I have listed some general points and good practices on the report writing. Please note that the list below is not exhaustive.

General points

1. Describe clearly the aim of the report and its task
2. Provide discussion on every result that you include. Does it make sense? What conclusions you can draw from the reported results?
3. Be precise when you try to justify the results. Do not just use phrases like "… good enough..", "..very slow…"
4. Avoid using screenshots that do not have any useful info.
5. Do not provide a description of the tutorial that you followed. If there is a specific step that gave you trouble or you think is important to mention, then do just that.
6. Have a conclusion section (task level/report). What did you see? (do not summarize what you did)
7. Pay attention to the structure of the report.
8. The report should be in pdf and you should submit your code only for the final report.
10. Provide names and group id on the first page of your report.
11. Think about adding graphs to make your points more clear.
12. Resources: Your system in the FPGA, and your Program (memory occupied)

Report 1 – Nice points to include

1. Think about the cache.
2. Think about the size of your program. Do you use dynamic memory allocation? When the process fail? (never during execution). Perform an offline analysis on the memory requirement of the various test cases to see whether you can fit the data in your system.
3. Finite precision. Float vs Double
4. Present your results using graphs and tables. It is better to argue your point in this way.
5. Always include a performance vs recourses graph in order to compare your various designs (present and past). Given that the aim is the same across the various tasks, it is interesting to see how the design of your system (moving functionality to dedicated HW) affects the performance and resource utilization of the system.

Report 2 - Nice points to include

1. Discuss the implementation of cos function
2. Discuss the impact of cache size

Report 3 - Nice points to include

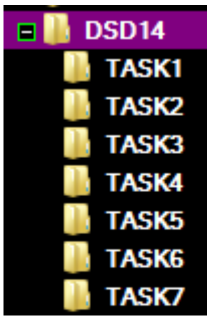
1. Comparison to previous implementations
2. Elaboration on the selection of the latency, achieved performance, area resources, achieved precision
3. Elaboration on the CORDIC architecture and justification on its architecture.
4. Elaboration on the final adder to complete the whole design (function evaluation in HW)

Code

Each group needs to upload the source files of their project for the most advanced design of each task, at the end of the project.

The source files should be organised with subdirectories according to the coursework tasks. Please include the following extensions:

.qpf - quartus project
.qsf - quartus settings
.sdc - timing constraints
.sopcinfo - system info
.qsys - system specification
.vhd - vhdl source files
.v - verilog source files
.qip - megawizard component
.tcl - script
.bdf - block diagrams
.bsf - symbols
.do - modelsim macro
.mpf - modelsim project
.c - C source files
.h - C include files
.bsp - bsp settings
.map
.hex
.bin
.cproject/.project - eclipse project defs



You can follow the instructions below to make a .ZIP file with the deliverables:

1. Make a copy of your work containing only the designs/software you want to submit. Make sure you create subfolders for each task, e.g. "task1", "task2", etc.;
2. start command line "cmd";
3. change drive and directory to be your DSD folder e.g. "e:" "cd /dsd14/";
4. execute the following command. It will copy your sources into a zip file;

```
zip dsd -rq . -i "*.h" "*.c" "*.bsp" "*.map" "*.qsf" "*.qpf" "*.sdc" "*.sopcinfo"
"*.qsys" "*.vhd" "*.v" "*.qip" "*.tcl" "*.bdf" "*.bsf" "*.do" "*.mpf" "*.*project"
"*.bin" "*.hex"
```

5. verify the zip file contains all the source files you want to submit for evaluation;
6. upload the file.

APPENDIX

Troubleshooting

Below there's a list of common problems and their solutions:

Problem	Solution
Unable to program the FPGA	Check whether the Programmer in Quartus has been properly set up for the device. It should be in JTAG mode through USB.
In Eclipse, you get the message: "Failed to execute: ./create-this-app –no-make"	This error may be generated by the Altera Complete Design Suite software version 10.0 and 10.0 SP1 when creating a software project by selecting **Nios II Application and BSP from Template** in the Nios II Software Build Tools (SBT) for Eclipse on Windows 7 using a non-administrator account. To work around this error when running on Windows 7, run the Nios II SBT for Eclipse as with administrator privileges. On the Windows Start menu, point to **All Programs**, point to **Altera**, point to **Nios II EDS**, right-click on **Nios II Software Build Tools for Eclipse**, and select **Run as administrator**.
Project fails to compile/run Analysis & Elaboration	There's an issue with the names of paths. To solve this you need to use a local directory instead of your home (H:) directory.
No output from printf("%f")	The small C library doesn't support printf("%f") you should use gcvt(), as illustrated in the handout.
Not enough memory or no output for on-chip your application	On-chip memory is very limited. You can't use functions from includes such as math.h. Work out your algorithm to do the same using basic arithmetic operators. Change the size of your on-chip memory to 40960-49152 bytes; Edit BSP and disable: C++, clean exit, exit; enable: small C, reduced drivers.
Simulation of the accelerator is OK, but it doesn't work with NIOS	You may have to simulate the complete system (not adviced). More details here: http://www.altera.com/support/examples/nios2/exm-simulating-niosii.html http://www.alterawiki.com/wiki/Simulating_Nios_II_Designs_with_Program_Memory_on_External_Flash_and_SSRAM
SDRAM settings for DE0 board	SDRAM settings: preset: custom; bits: 16; cs: 1; banks: 4; row: 12; col: 8; cas: 3; init: 2; refresh: 15.625; delay: 100; t_rfc: 70; t_rp: 20; t_rcd: 20; t_ac: 5.5; t_wr: 14;
It takes too long from generation of the system to running an application in NIOS	To accelerate the process, **disable (in Qsys) all components that are not being used in your system** (e.g. on-chip memory and led_pio). Less components means processing less time: generating the system in Qsys, compiling the design in Quartus II, generating the BSP and compiling less drivers with your application. :). Do not forget to re-generate the memory address space.
Custom Instruction	Figure on page 25: Your data are passed through dataa and datab (32 bits), not through ports a and b (5 bits).
Name of Components in Qsys and Quartus	Remove "_0" from the default name given by Qsys/Quartus. Don't use "_" and a number as a suffix for the names of your units.

DE0 Board Programming

The following steps describe how to program the DE0 board.

1. On Quartus II v.13.1. Select: **Tools → Programmer**
2. Add your project and run!
3. Remember to always stop the NIOS II application in EDS, otherwise you may lose ability to reprogram the board and you'll have to power cycle it before using it again.

Quartus II Tutorial

Quartus is the application that maps your design to a particular FPGA device. More information on Quartus can be found in http://www.altera.com/literature/manual/archives/intro_to_quartus2.pdf.

Modelsim Tutorial

Modelsim is the application where you can run a simulation of your design before deploying into the FPGA, saving development time.

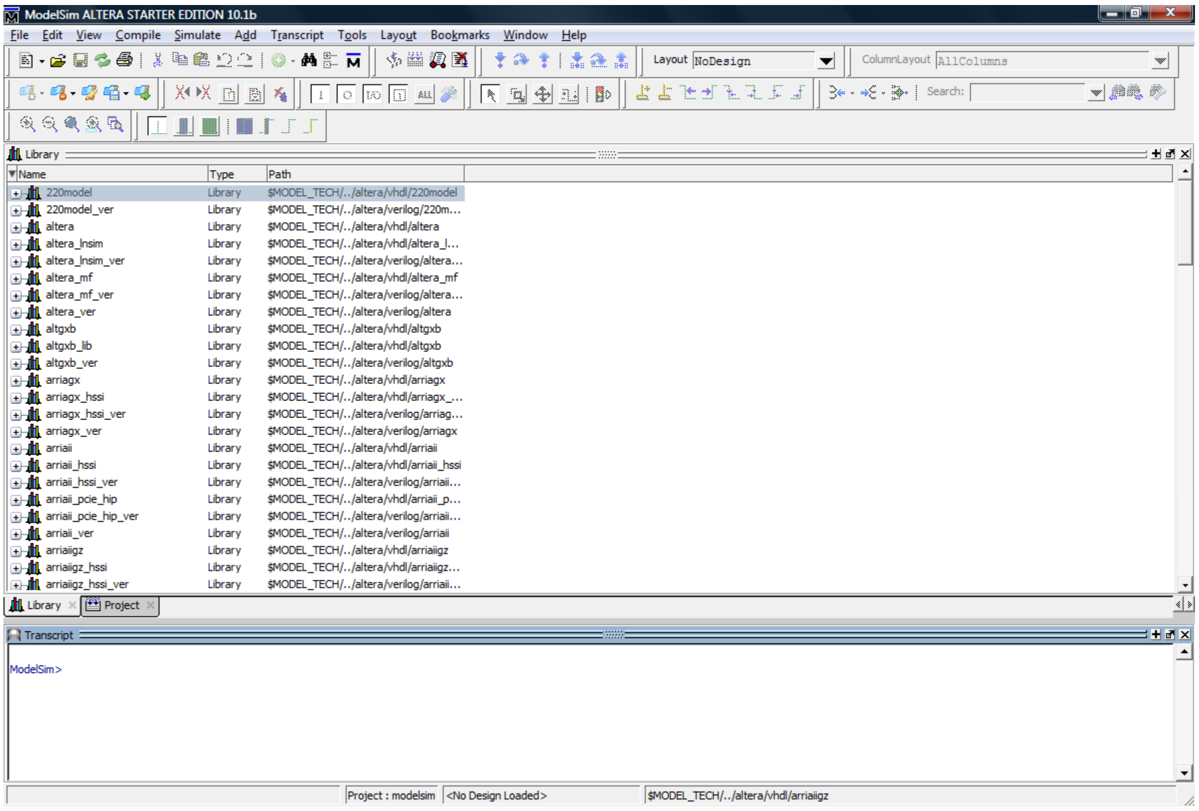
If you chose to use Verilog please refer to the document **modelsim_tutorial_verilog.pdf** available in BlackBoard.

You need to compile the Altera libraries to be able to run the functional simulation in Modelsim:

📖 http://www.altera.com/literature/hb/qts/qts_qii5v3.pdf
http://quartushelp.altera.com/10.0/mergedProjects/eda/simulation/modelsim/eda_pro_msimfull_compile.htm

INTERFACE

Below there's a figure with the aspect of Modelsim after you start it. On the top you have the menus and toolbars. In the middle you have two panels. One has the list of libraries included, and the other has the project's dependencies. In the bottom you have the console, where you can write commands and read their output.



ModelSim ALTERA STARTER EDITION 10.1b
File Edit View Compile Simulate Add Transcript Tools Layout Bookmarks Window Help
Layout NoDesign
ColumnLayout AllColumns
Search:
Library
Library
Project
Transcript

Name	Type	Path
220model	Library	$MODEL_TECH/../altera/vhdl/220model
220model_ver	Library	$MODEL_TECH/../altera/verilog/220m...
altera	Library	$MODEL_TECH/../altera/vhdl/altera
altera_lnsim	Library	$MODEL_TECH/../altera/vhdl/altera_l...
altera_lnsim_ver	Library	$MODEL_TECH/../altera/verilog/altera...
altera_mf	Library	$MODEL_TECH/../altera/vhdl/altera_mf
altera_mf_ver	Library	$MODEL_TECH/../altera/verilog/altera...
altera_ver	Library	$MODEL_TECH/../altera/verilog/altera
altgxb	Library	$MODEL_TECH/../altera/vhdl/altgxb
altgxb_lib	Library	$MODEL_TECH/../altera/vhdl/altgxb
altgxb_ver	Library	$MODEL_TECH/../altera/verilog/altgxb
arriagx	Library	$MODEL_TECH/../altera/vhdl/arriagx
arriagx_hssi	Library	$MODEL_TECH/../altera/vhdl/arriagx_...
arriagx_hssi_ver	Library	$MODEL_TECH/../altera/verilog/arriag...
arriagx_ver	Library	$MODEL_TECH/../altera/verilog/arriagx
arriaii	Library	$MODEL_TECH/../altera/vhdl/arriaii
arriaii_hssi	Library	$MODEL_TECH/../altera/vhdl/arriaii_hssi
arriaii_hssi_ver	Library	$MODEL_TECH/../altera/verilog/arriaii...
arriaii_pcie_hip	Library	$MODEL_TECH/../altera/vhdl/arriaii_p...
arriaii_pcie_hip_ver	Library	$MODEL_TECH/../altera/verilog/arriaii...
arriaii_ver	Library	$MODEL_TECH/../altera/verilog/arriaii
arriaiigz	Library	$MODEL_TECH/../altera/vhdl/arriaiigz
arriaiigz_hssi	Library	$MODEL_TECH/../altera/vhdl/arriaiigz...
arriaiigz_hssi_ver	Library	$MODEL_TECH/../altera/verilog/arriaii...

```
ModelSim>
```

Project : modelsim
<No Design Loaded>
$MODEL_TECH/../altera/vhdl/arriaiigz

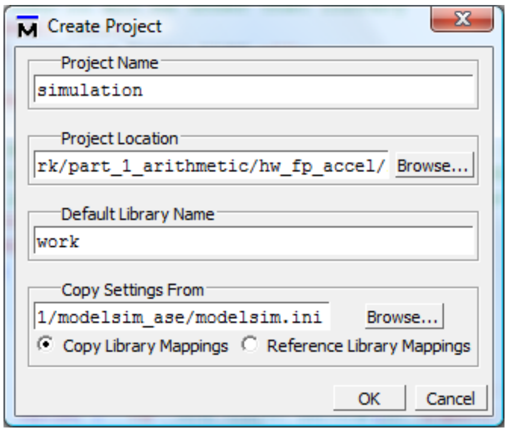
Depending on the task being done, Modelsim will change its interface to facilitate interaction or add extra functionality.

CREATE PROJECT

To create a simulation of a design you need to create a project, and add all dependencies to the project.

File→New→Project

Specify the name of your simulation project and its location. Modelsim's project files have .mpf extension.



Project Name: simulation

Project Location: rk/part_1_arithmetic/hw_fp_accel/

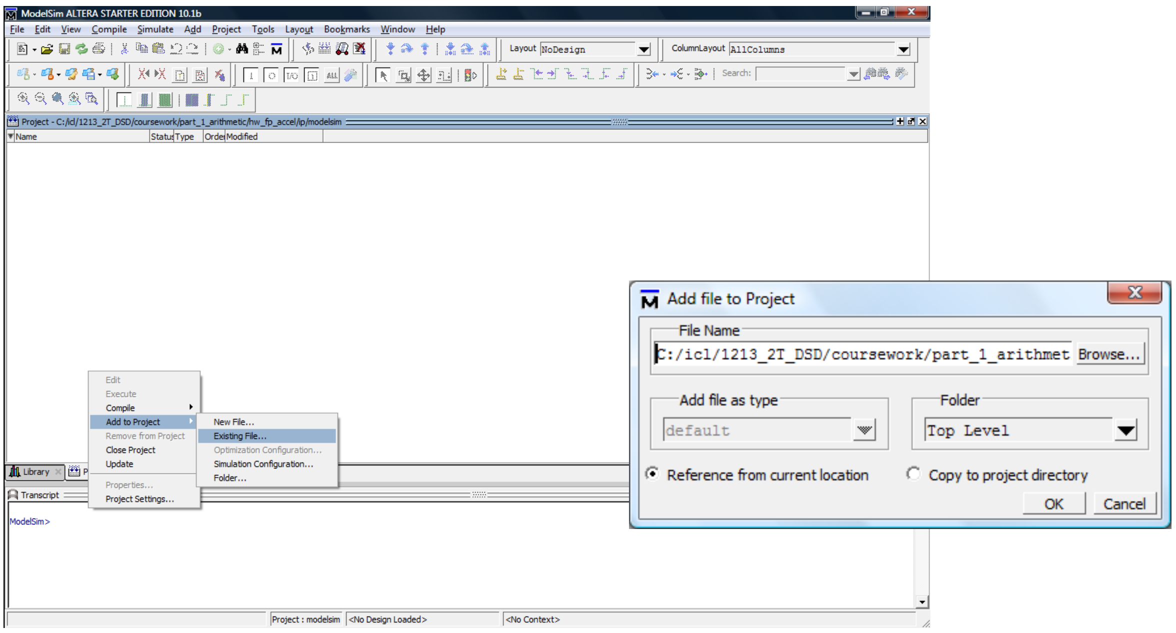
Default Library Name: work

Copy Settings From: 1/modelsim_ase/modelsim.ini

[✓ Copy Library Mappings] Reference Library Mappings

At this point you should have all source files generated by Quartus. All you need to do it to add the VHD source files, from your IP folder, to your project as a reference.

Right-click on the project panel→Add to Project→Existing File...



Project - C:/icl/1213_2T_DSD/coursework/part_1_arithmetic/hw_fp_accel/ip/modelsim

Layout: NoDesign
ColumnLayout: AllColumns

Add file to Project

File Name: C:/icl/1213_2T_DSD/coursework/part_1_arithmet

Add file as type: default

Folder: Top Level

[✓ Reference from current location] Copy to project directory

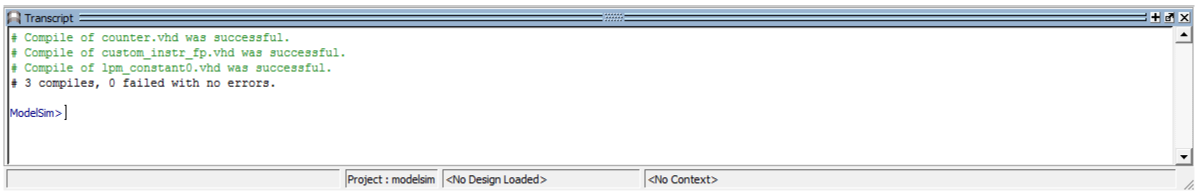
Project : modelsim <No Design Loaded> <No Context>

COMPILE

Before you're able to run a simulation you need to compile your project first.

Compile all of your source files:

You should get no errors:



```
# Compile of counter.vhd was successful.
# Compile of custom_instr_fp.vhd was successful.
# Compile of lpm_constant0.vhd was successful.
# 3 compiles, 0 failed with no errors.

ModelSim>
```

Project : modelsim <No Design Loaded> <No Context>

SIMULATION

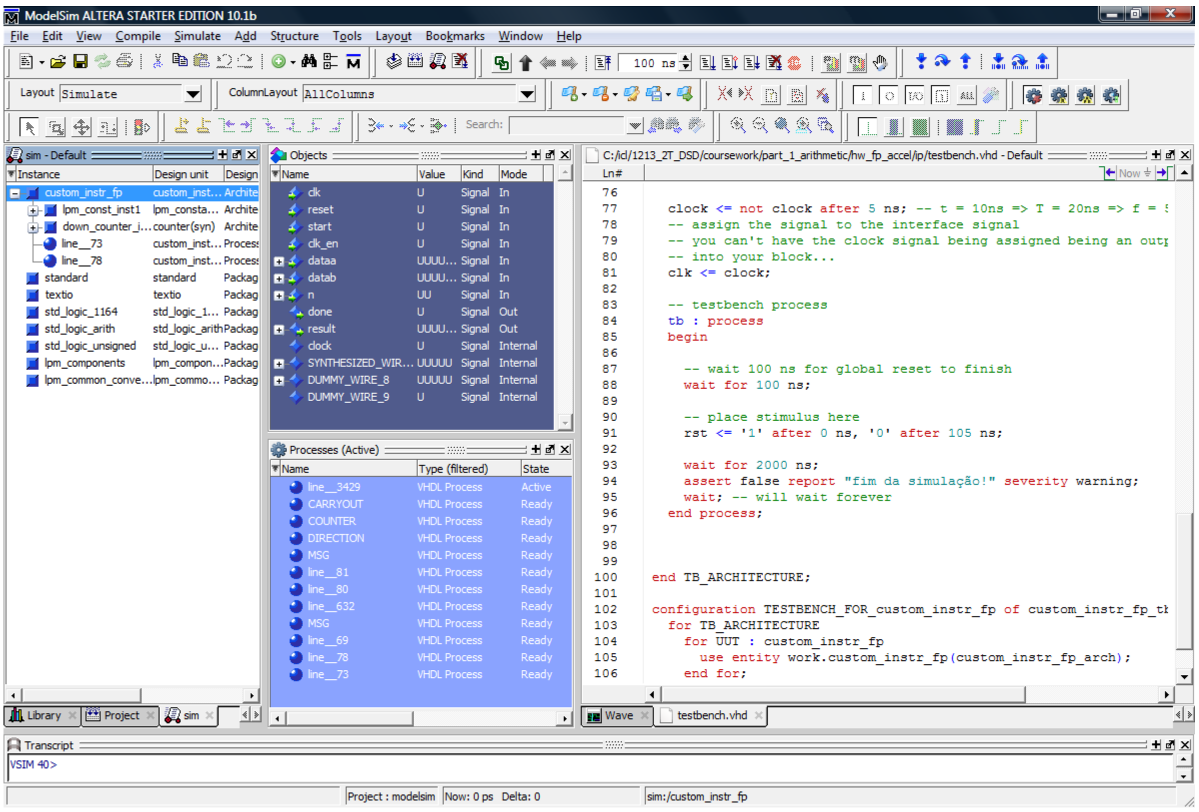
To run a simulation you need to specify a stimulus for your design, so you can check it works accordingly.

There are two ways of doing it: graphical interface or using a *testbench* and macro files.

GUI

Graphical is more intuitive to start doing small simulations, but it requires that you manually assign the stimulus to every signal.

After you click Simulate, Modelsim will change the middle panels to something similar to the figure below:


ModelSim ALTERA STARTER EDITION 10.1b
File Edit View Compile Simulate Add Structure Tools Layout Bookmarks Window Help
Library
Project
sim
Wave
testbench.vhd

Layout: Simulate — ColumnLayout: AllColumns — 100 ns

sim - Default

Instance	Design unit	Design
custom_instr_fp	custom_inst…	Archite
lpm_const_inst1	lpm_consta…	Archite
down_counter_i…	counter(syn)	Archite
line__73	custom_inst…	Process
line__78	custom_inst…	Process
standard	standard	Packag
textio	textio	Packag
std_logic_1164	std_logic_1…	Packag
std_logic_arith	std_logic_arith	Packag
std_logic_unsigned	std_logic_u…	Packag
lpm_components	lpm_compon…	Packag
lpm_common_conve…	lpm_commo…	Packag

Objects

Name	Value	Kind	Mode
clk	U	Signal	In
reset	U	Signal	In
start	U	Signal	In
clk_en	U	Signal	In
dataa	UUUU…	Signal	In
datab	UUUU…	Signal	In
n	UU	Signal	In
done	U	Signal	Out
result	UUUU…	Signal	Out
clock	U	Signal	Internal
SYNTHESIZED_WIR…	UUUUU	Signal	Internal
DUMMY_WIRE_8	UUUUU	Signal	Internal
DUMMY_WIRE_9	U	Signal	Internal

Processes (Active)

Name	Type (filtered)	State
line__3429	VHDL Process	Active
CARRYOUT	VHDL Process	Ready
COUNTER	VHDL Process	Ready
DIRECTION	VHDL Process	Ready
MSG	VHDL Process	Ready
line__81	VHDL Process	Ready
line__80	VHDL Process	Ready
line__632	VHDL Process	Ready
MSG	VHDL Process	Ready
line__69	VHDL Process	Ready
line__78	VHDL Process	Ready
line__73	VHDL Process	Ready

C:/icl/1213_2T_DSD/coursework/part_1_arithmetic/hw_fp_accel/ip/testbench.vhd - Default

```

    clock <= not clock after 5 ns; -- t = 10ns => T = 20ns => f = 5
    -- assign the signal to the interface signal
    -- you can't have the clock signal being assigned being an outp
    -- into your block...
    clk <= clock;

    -- testbench process
    tb : process
    begin

      -- wait 100 ns for global reset to finish
      wait for 100 ns;

      -- place stimulus here
      rst <= '1' after 0 ns, '0' after 105 ns;

      wait for 2000 ns;
      assert false report "fim da simulação!" severity warning;
      wait; -- will wait forever
    end process;



  end TB_ARCHITECTURE;

  configuration TESTBENCH_FOR_custom_instr_fp of custom_instr_fp_tb
    for TB_ARCHITECTURE
      for UUT : custom_instr_fp
        use entity work.custom_instr_fp(custom_instr_fp_arch);
      end for;
```

Transcript

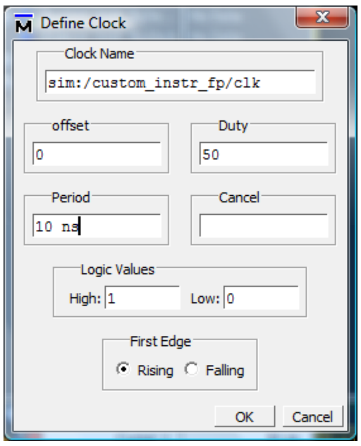
```
VSIM 40>
```

Project : modelsim — Now: 0 ps Delta: 0 — sim:/custom_instr_fp

On the left side you have the hierarchy of units belonging to your hardware design. In the middle, the signals for the unit selected on the left, and the code editor on the right. If you don't have a "wave" panel behind the editor, click on View→Wave.

Select the wave panel and drag the top unit of your design to the wave panel. Their signals will appear on the left side of the wave diagram.

Right-click on the clock signal and select "Clock", specify 10 ns clock period and run the simulation for 100 ns



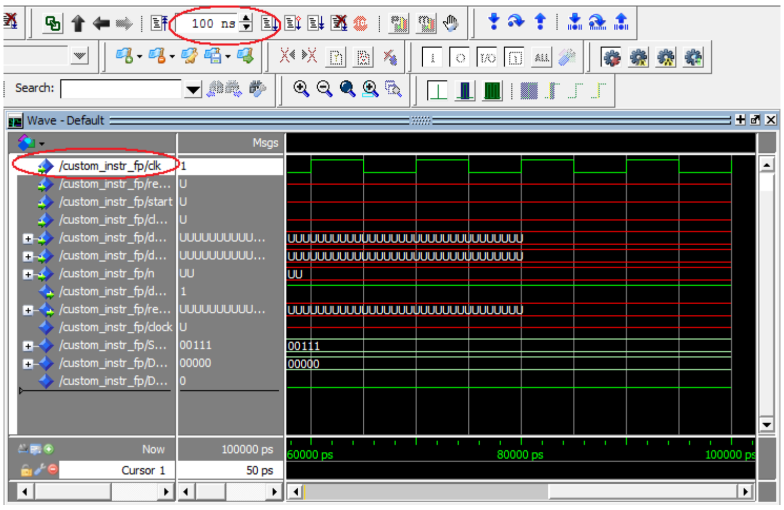
Clock Name: sim:/custom_instr_fp/clk

offset: 0 — Duty: 50

Period: 10 ns — Cancel:

Logic Values — High: 1 — Low: 0

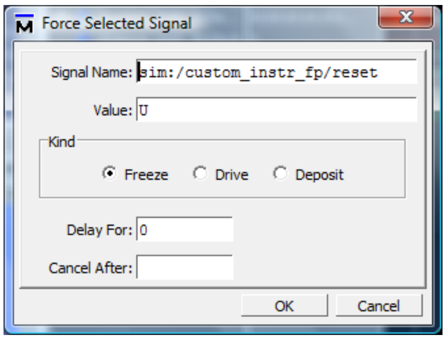
First Edge: [✓ Rising] Falling



100 ns

	Msgs
/custom_instr_fp/clk	1
/custom_instr_fp/re…	U
/custom_instr_fp/start	U
/custom_instr_fp/cl…	U
/custom_instr_fp/d…	UUUUUUUUUU…
/custom_instr_fp/d…	UUUUUUUUUU…
/custom_instr_fp/n	UU
/custom_instr_fp/d…	1
/custom_instr_fp/re…	UUUUUUUUUU…
/custom_instr_fp/clock	U
/custom_instr_fp/S…	00111
/custom_instr_fp/D…	00000
/custom_instr_fp/D…	0
Now	100000 ps
Cursor 1	50 ps

For all the remaining input stimulus signals you need to force their transition (when and which value) and re-run the simulation.



Signal Name: sim:/custom_instr_fp/reset
Value: U
Kind: [✓ Freeze] Drive Deposit
Delay For: 0
Cancel After:

TESTBENCH AND MACRO FILES

An easier way to setup the simulation is to use a *testbench*. A *testbench* is basically a VHDL entity for simulation, where your hardware design is instantiated. It supports VHDL statements specific for simulation, which you can't synthesize, namely delays, which are useful to program stimulus.

In the coursework material you will find a file named **testbench.vhd**. It has the stimulus for interfacing NIOS II with a hardware block with multiple custom instructions.

Copy it to your ./ip folder, add it to your project and re-compile it. You may need to change names of signals in the file to match your design.

Once you run the simulation you'll verify that your design is now the UUT (Unit Under Test). Drag it to an empty wave window. Run the simulation for 300 ns. You'll find that some signals are set up, namely the clock and reset signals.

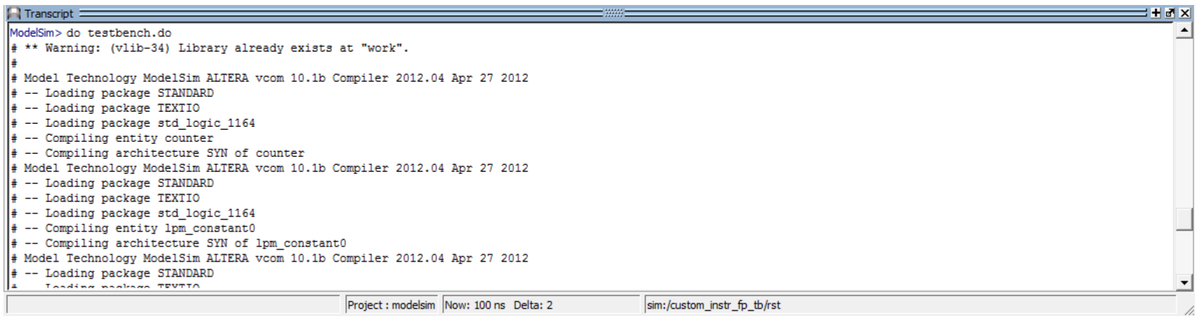
To avoid having to define the wave diagram for every simulation, it is possible to automate the process running a macro: testbench.do

```
alias t "do testbench.do"

vlib work

vcom -work work counter.vhd
vcom -work work lpm_constant0.vhd
vcom -work work lpm_constant1.vhd
vcom -work work lpm_mux32.vhd
vcom -work work custom_instr_fp.vhd
vcom -work work testbench.vhd

vcom -work work fp_add_sub.vhd
vcom -work work fp_mult.vhd
vcom -work work mw_mux.vhd

vsim -t ps work.custom_instr_fp_tb

add wave -position end  sim:/custom_instr_fp_tb/clk
add wave -position end  sim:/custom_instr_fp_tb/rst
add wave -position end  sim:/custom_instr_fp_tb/enable
add wave -position end  sim:/custom_instr_fp_tb/start
add wave -position end  sim:/custom_instr_fp_tb/d
add wave -position end  sim:/custom_instr_fp_tb/clock
add wave -position end  sim:/custom_instr_fp_tb/UUT/clk
add wave -position end  sim:/custom_instr_fp_tb/UUT/reset
add wave -position end  sim:/custom_instr_fp_tb/UUT/start
add wave -position end  sim:/custom_instr_fp_tb/UUT/clk_en
add wave -position end  sim:/custom_instr_fp_tb/UUT/dataa
add wave -position end  sim:/custom_instr_fp_tb/UUT/datab
add wave -position end  sim:/custom_instr_fp_tb/UUT/n
add wave -position end  sim:/custom_instr_fp_tb/UUT/done
add wave -position end  sim:/custom_instr_fp_tb/UUT/result
add wave -position end  sim:/custom_instr_fp_tb/UUT/start_stop_unit
add wave -position end  sim:/custom_instr_fp_tb/UUT/unit_active

run
```

This macro has Modelsim commands to compile the VHDL source files (use “vcom” for VHDL, “vlog” for Verilog), run the simulation and place the signals in the wave window. To execute it, write the following statement on the console window:

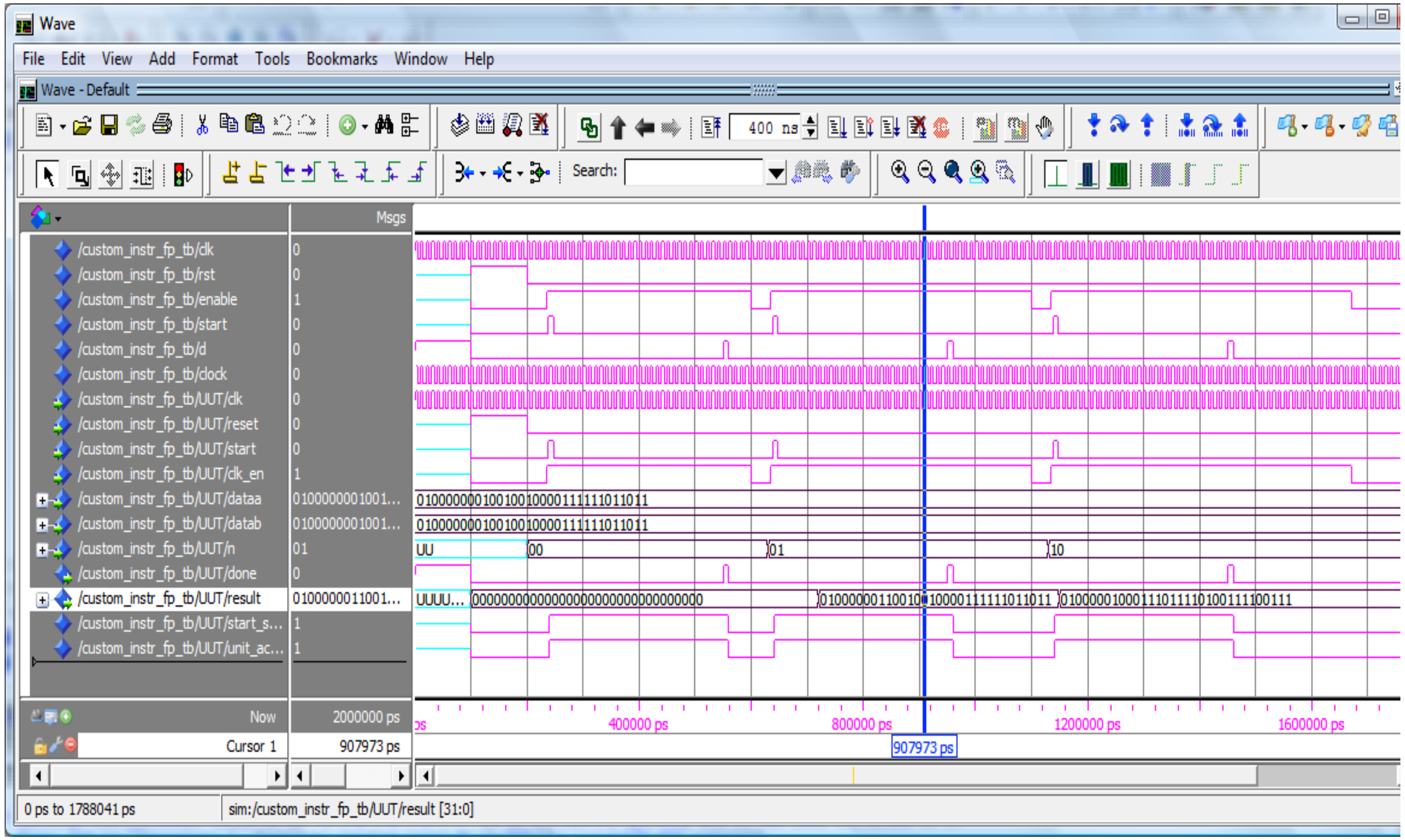
do testbench.do

These files cover the basic simulation functionalities and you can adapt them to simulate your hardware blocks.



```
Transcript
ModelSim> do testbench.do
# ** Warning: (vlib-34) Library already exists at "work".
#
# Model Technology ModelSim ALTERA vcom 10.1b Compiler 2012.04 Apr 27 2012
# -- Loading package STANDARD
# -- Loading package TEXTIO
# -- Loading package std_logic_1164
# -- Compiling entity counter
# -- Compiling architecture SYN of counter
# Model Technology ModelSim ALTERA vcom 10.1b Compiler 2012.04 Apr 27 2012
# -- Loading package STANDARD
# -- Loading package TEXTIO
# -- Loading package std_logic_1164
# -- Compiling entity lpm_constant0
# -- Compiling architecture SYN of lpm_constant0
# Model Technology ModelSim ALTERA vcom 10.1b Compiler 2012.04 Apr 27 2012
# -- Loading package STANDARD
# -- Loading package TEXTIO
Project : modelsim   Now: 100 ns  Delta: 2   sim:/custom_instr_fp_tb/rst
```

Below the figure shows the wave diagram for the simulation of the Floating-Point unit, implementing addition, subtraction and multiplication operations.



```
/custom_instr_fp_tb/clk              0
/custom_instr_fp_tb/rst              0
/custom_instr_fp_tb/enable           1
/custom_instr_fp_tb/start            0
/custom_instr_fp_tb/d                0
/custom_instr_fp_tb/clock            0
/custom_instr_fp_tb/UUT/clk          0
/custom_instr_fp_tb/UUT/reset        0
/custom_instr_fp_tb/UUT/start        0
/custom_instr_fp_tb/UUT/clk_en       1
/custom_instr_fp_tb/UUT/dataa        0100000001001...   01000000010010010000111111011011
/custom_instr_fp_tb/UUT/datab        0100000001001...   01000000010010010000111111011011
/custom_instr_fp_tb/UUT/n            01                 UU  00  01  10
/custom_instr_fp_tb/UUT/done         0
/custom_instr_fp_tb/UUT/result       0100000011001...   UUUU...  00000000000000000000000000000000  01000000110010010000111111011011  01000001000111011110100111100111
/custom_instr_fp_tb/UUT/start_s...   1
/custom_instr_fp_tb/UUT/unit_ac...   1

Now        2000000 ps
Cursor 1   907973 ps     907973 ps

0 ps to 1788041 ps    sim:/custom_instr_fp_tb/UUT/result [31:0]
```

Application Profiling

In the quest to accelerate complex applications, it may not be obvious which instructions take most of the processor's time. To measure the performance of parts of your application you can make use of an external profiler or measure the time require. Altera provides an application note specific on this subject. You can find it here:

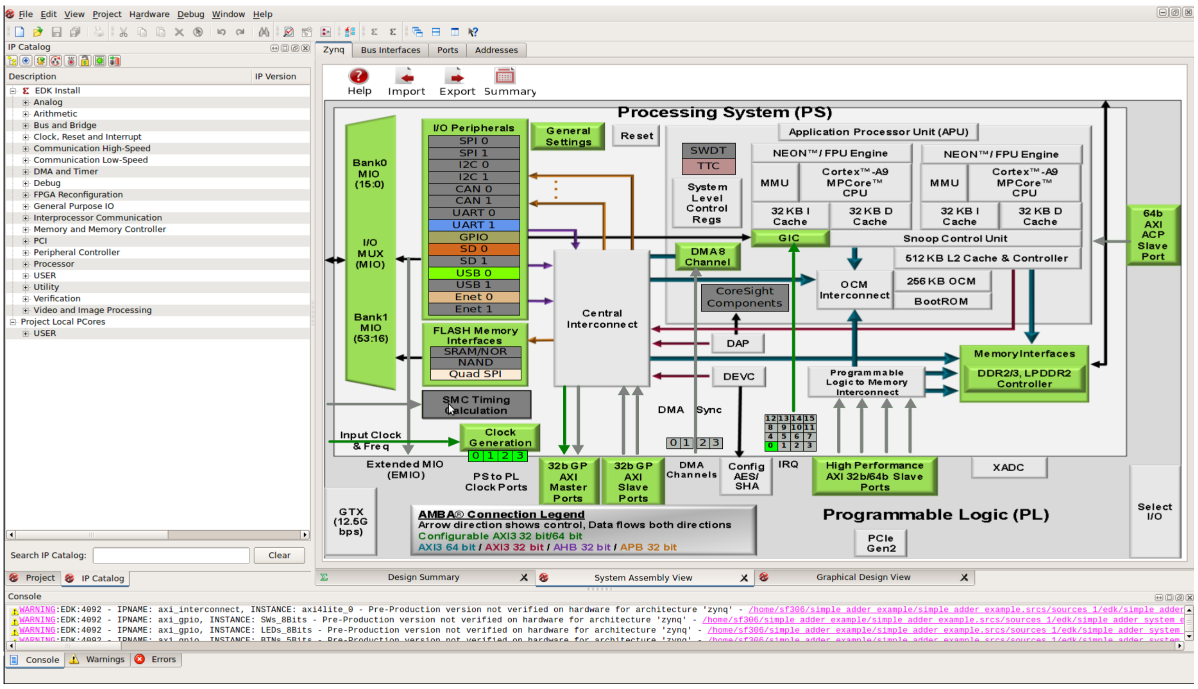
📖 http://www.altera.com/literature/an/an391.pdf

Alternatives to NIOS II/Altera

Embedded systems with soft/hard processors and reconfigurable logic are common. Similar to Altera (NIOS II) there are alternatives, namely Zynq from Xilinx. The Zynq system has a hard processor occupying a region of the device, which cannot be used for reconfigurable logic, but on the other hand the processor has enough computational power to run Ubuntu on it.

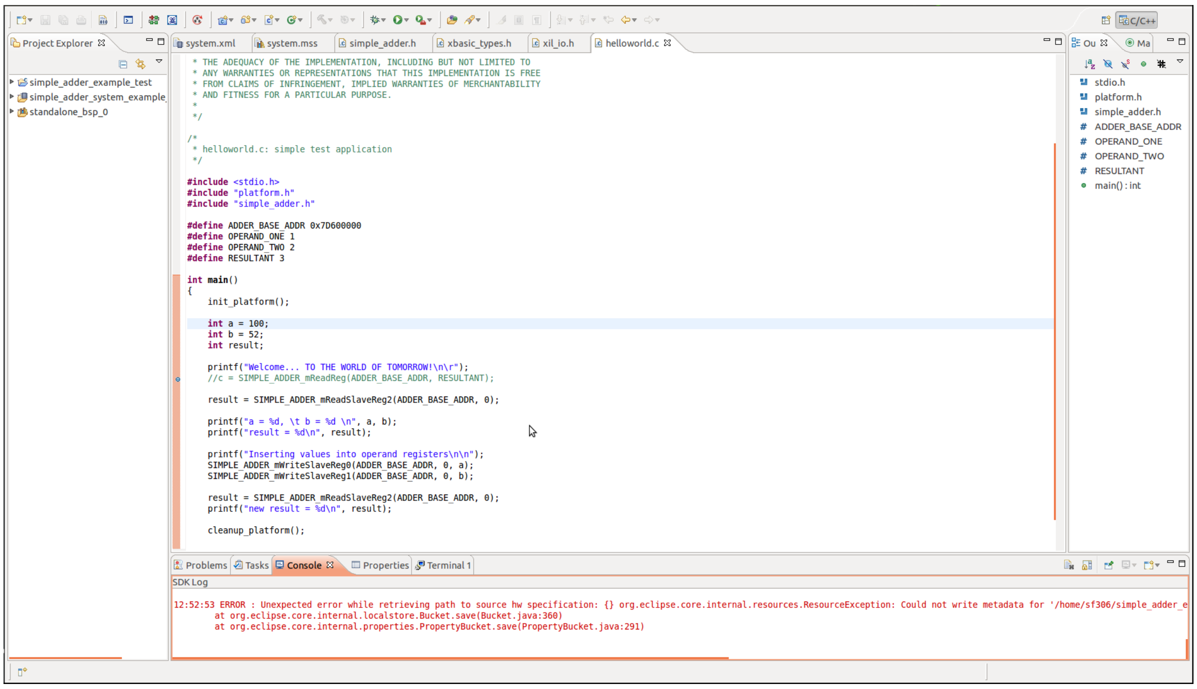
Here are screenshots of the development environments for hardware (XPS) and software (SDK). You'll find them similar to the tools you're using in this coursework.

XPS



```
WARNING:EDK:4092 - IPNAME: axi_interconnect, INSTANCE: axi4lite_0 - Pre-Production version not verified on hardware for architecture 'zynq' - /home/sf306/simple adder example/simple adder example.srcs/sources 1/edk/simple adder
WARNING:EDK:4092 - IPNAME: axi_gpio, INSTANCE: SWs_8Bits - Pre-Production version not verified on hardware for architecture 'zynq' - /home/sf306/simple adder example/simple adder example.srcs/sources 1/edk/simple adder system e
WARNING:EDK:4092 - IPNAME: axi_gpio, INSTANCE: LEDs_8Bits - Pre-Production version not verified on hardware for architecture 'zynq' - /home/sf306/simple adder example/simple adder example.srcs/sources 1/edk/simple adder system
WARNING:EDK:4092 - IPNAME: axi_gpio, INSTANCE: BTNs_5Bits - Pre-Production version not verified on hardware for architecture 'zynq' - /home/sf306/simple adder example/simple adder example.srcs/sources 1/edk/simple adder system
```

SDK



```
 * THE ADEQUACY OF THE IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO
 * ANY WARRANTIES OR REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE
 * FROM CLAIMS OF INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.
 *
 */

/*
 * helloworld.c: simple test application
 */

#include <stdio.h>
#include "platform.h"
#include "simple_adder.h"

#define ADDER_BASE_ADDR 0x7D600000
#define OPERAND_ONE 1
#define OPERAND_TWO 2
#define RESULTANT 3

int main()
{
    init_platform();

    int a = 100;
    int b = 52;
    int result;

    printf("Welcome... TO THE WORLD OF TOMORROW!\n\r");
    //c = SIMPLE_ADDER_mReadReg(ADDER_BASE_ADDR, RESULTANT);

    result = SIMPLE_ADDER_mReadSlaveReg2(ADDER_BASE_ADDR, 0);

    printf("a = %d, \t b = %d \n", a, b);
    printf("result = %d\n", result);

    printf("Inserting values into operand registers\n\n");
    SIMPLE_ADDER_mWriteSlaveReg0(ADDER_BASE_ADDR, 0, a);
    SIMPLE_ADDER_mWriteSlaveReg1(ADDER_BASE_ADDR, 0, b);

    result = SIMPLE_ADDER_mReadSlaveReg2(ADDER_BASE_ADDR, 0);
    printf("new result = %d\n", result);

    cleanup_platform();
```

```
12:52:53 ERROR : Unexpected error while retrieving path to source hw specification: {} org.eclipse.core.internal.resources.ResourceException: Could not write metadata for '/home/sf306/simple_adder_e
        at org.eclipse.core.internal.localstore.Bucket.save(Bucket.java:360)
        at org.eclipse.core.internal.properties.PropertyBucket.save(PropertyBucket.java:291)
```

*** end of document ***