Imperial College
London

DIGITAL SYSTEM DESIGN

COURSEWORK

TABLE OF CONTENTS

Requirements to complete this coursework:

Quartus II (v.12);

Nios II Embedded Design Suite (v.12);

Modelsim-Altera (or ActiveHDL);

Matlab (or Python);

Terasic DE2-70 board;

Coursework source files including DE2-70 documentation and Control Panel application;

References from Altera.

Legend:

ⓘ Important information.

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

🕷 Debug hints

💰 Optional work rewarded with extra marks. It has the purpose to encourage discussing further implementation features or options.

INTRODUCTION

In this coursework you will acquire experience in the design of digital systems targeting intensive arithmetic and streaming applications.

Many applications require large amounts of calculations to be performed. Most of the digital systems available have one or more dedicated units to do basic arithmetic operations: addition, subtraction, multiplication and division. From the above list of operations, multiplication and division are the most area and time consuming operations. They are the performance bottleneck in most computational systems.

In this coursework we will start running applications on a generic processing unit (CPU) 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. The FPGA is on a development kit (DE2-70) 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 specifying and program digital systems based on FPGA. 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.cn/literature/hb/qts/qsys_intro.pdf .

ⓘ Please read this document carefully 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. Make sure you read and understand them.

PART I – ARITHMETIC APPLICATIONS

In the first part of the coursework we are going to see how dedicated hardware can be applied to a general purpose digital system to accelerate calculations. These applications are usually characterized by having to compute the same sequence of operations often on large amounts of data. The main goal of hardware accelerators, in this case, is to reduce the workload on the processor.

NIOS II Setup

Altera, the manufacturer of the target FPGA device, 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.

🛠 Know how to setup a system with a NIOS II processor, and start building your project. Please follow the NIOS II tutorial provided by Altera, and use the project template available on the DSD coursework page. It will help you setting up and interacting with a system with NIOS II processor. http://www.altera.com/literature/tt/tt_nios2_hardware_tutorial.pdf

The above tutorial will help you to implement a NIOS II processor on an FPGA using the on-chip memory. The generated system has an ALU that supports fixed-point arithmetic, where all the arithmetic operators are implemented using LUTs.

The FPGA on the DE2-70 board is the Cyclone II EP2C70F896C6.

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 can be determined using the following statements in C code:

```
  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(((double)exec_t2-(double)exec_t1) / alt_ticks_per_second(), 10, buf);
  alt_putstr(" proc time = ");   alt_putstr(buf);    alt_putstr(" seconds \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.

Determinant of a Square Matrix

Now that you have your system running, we can start using it to make computations. The operation that you are going to implement is a function that calculates the determinant of a matrix.

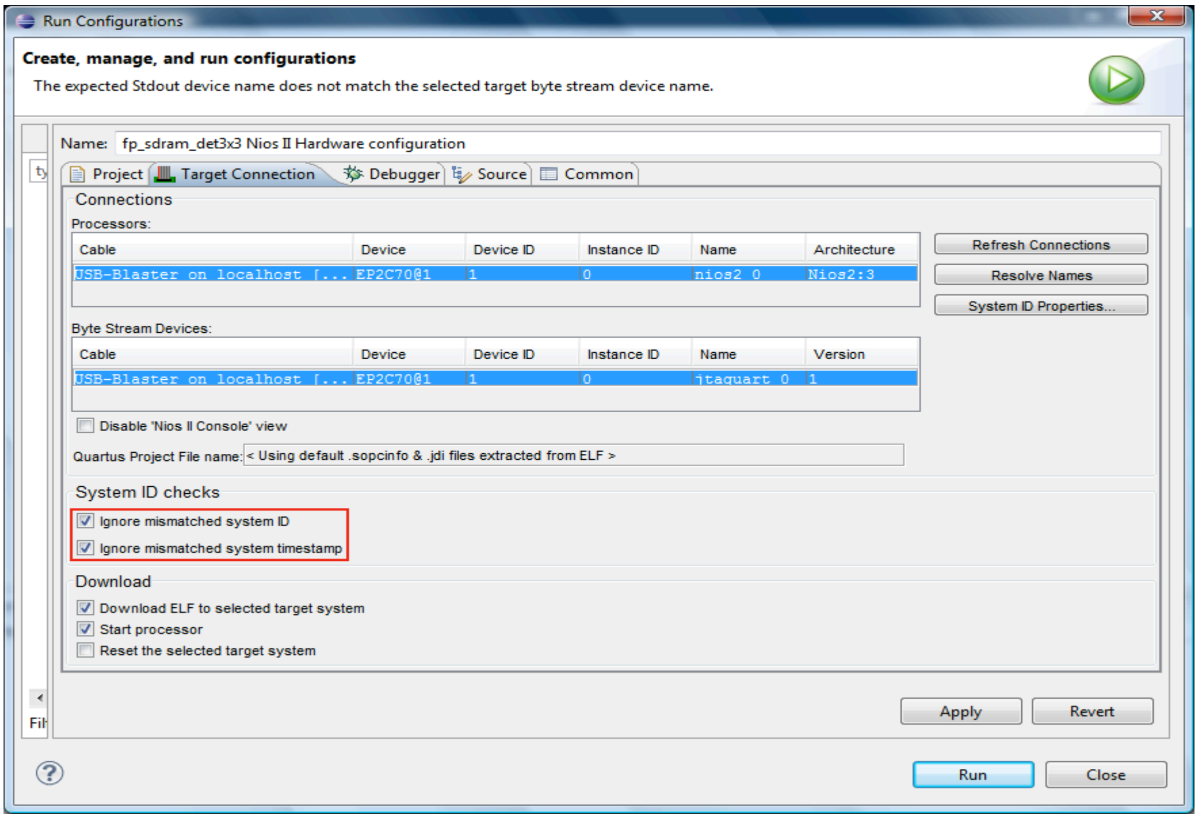
https://en.wikipedia.org/wiki/Determinant

Write a new software application for your NIOS II system to compute the determinant of a 3x3 floating-point (single precision) matrix. Detailed information about programming NIOS II in EDS is offered here: http://www.altera.com/literature/hb/nios2/n2sw_nii52017.pdf

How long does NIOS II take to compute the determinant?

If your determinant calculation is faster than 1us you may want to execute the function many times using a for loop.

To debug your application running on NIOS II use instead of . Make use of 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. If you encounter problems launching your application on the FPGA you may need to tick the boxes to ignore the system's timestamp and/or system's ID.


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

Good to know:

One of the parameters of NIOS II is the size of the cache. This parameter should be chosen carefully as it impacts the performance of the CPU. So far NIOSII was configured with 2k byte of cache. Since the data and instructions are stored on-chip, the presence and the size of cache won't impact the system's performance. Later, in more complex designs you may have to assess the correct cache size for your application.

Storing Program and Data on External Memories

The system that you created to perform the determinant calculation has its application stored 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 DE2-70 board there are different types of memories, including SDRAM memory. The main benefit of using it, 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 performance, when compared to on-chip memory.

Good to know:

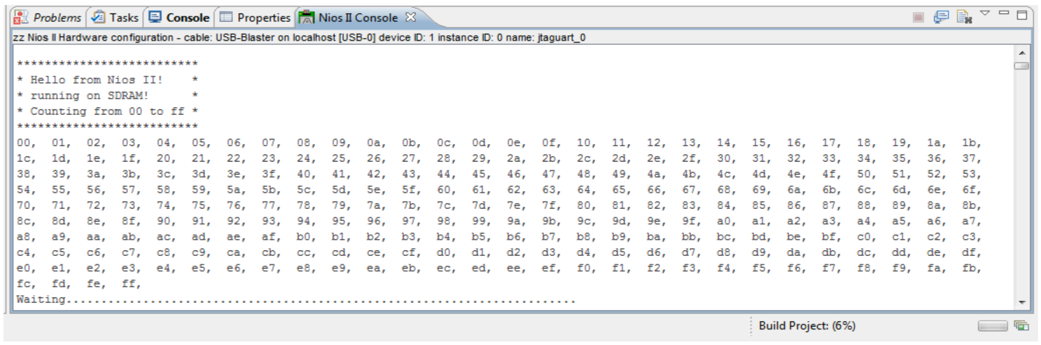
Parameters for the SDRAM controller to work with the SDRAM on the DE2-70:

```
Presets: custom, Data width: 32, Chip select: 1, Banks: 4, Address: 13+9, no
controller shares, no memory model, CAS latency: 3, Initialization: 2, refresh:
7.8125, Power-up: 200, t_rfc: 70, t_rp: 20, t_rcd: 20, t_ac: 5.5, t_wr: 14.
```

Follow the guide on ftp://ftp.altera.com/up/pub/Altera_Material/12.0/Tutorials/Verilog/DE2-70/Using_the_SDRAM.pdf to replace on-chip memory in your system with SDRAM. Run the same application you created before and report its performance. Whenever you introduce changes in Qsys you will need to re-generate, re-synthesize and re-configure your system.

Please note that SDRAM is very sensitive to clock variations. It is recommended that you use an external PLL in your design. You can insert the PLL from MegaWizard and place it on your Quartus II top-level entity. You may have to “fine-tune” the PLL time shift to a value between -2.62ns and -2.50ns (recommended -2.55ns). You can read more about it 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.



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...........................................................................
```

Now that you have more memory available in your system, update your application to compute the determinant for larger matrices, such as 6x6, 8x8, 10x10 and 20x20.

How long does it take to compute the determinant for each matrix? What is the size of your NIOS II application for the different matrices sizes?

You can read the size of your NIOS II application by inspecting the generated map file or the console after it is sent to NIOS. We are interested in comparing the size of the application for the different implementations. As we move the computations from software to hardware, the size of the application may be affected.

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 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!

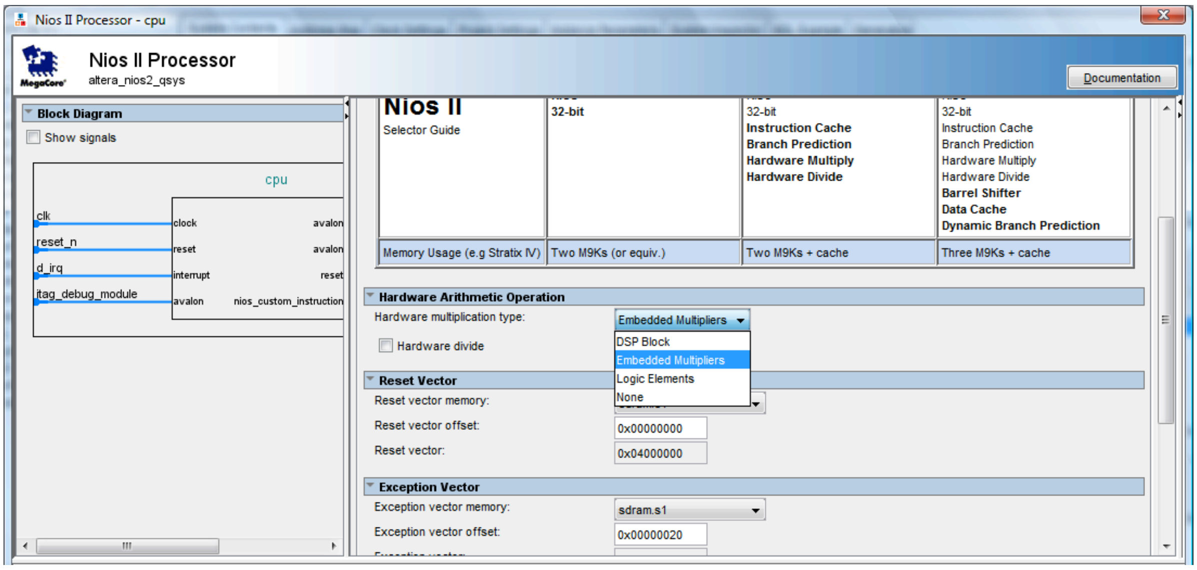
- Stop the execution of your application before trying to re-program the FPGA, or you may loose connection to it.

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 II http://www.altera.com/literature/lit-cyc2.jsp has dedicated hardware multiplier blocks that can be used to do fixed-point (or the mantissa in floating-point) multiplications faster. Still, the remaining floating-point arithmetic has to be emulated (signs, exponents, rounding and packing).

Enable support for embedded multipliers in your NIOS II processor. Re-generate and re-synthesize your system, repeat the same experiments and report your execution time and code size.

(Optional) LUT-based multipliers are often used on FPGAs due to the flexibility of their word-length configuration, allowing having faster multipliers at the expense of computing fewer bits than generic fixed word-length 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 (9x9 bits). **Bonus mark** (2%) if you determine by how much you would have lost in terms of your system's performance if you were using the LUT-based multipliers?



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

Hardware Floating-Point Multipliers

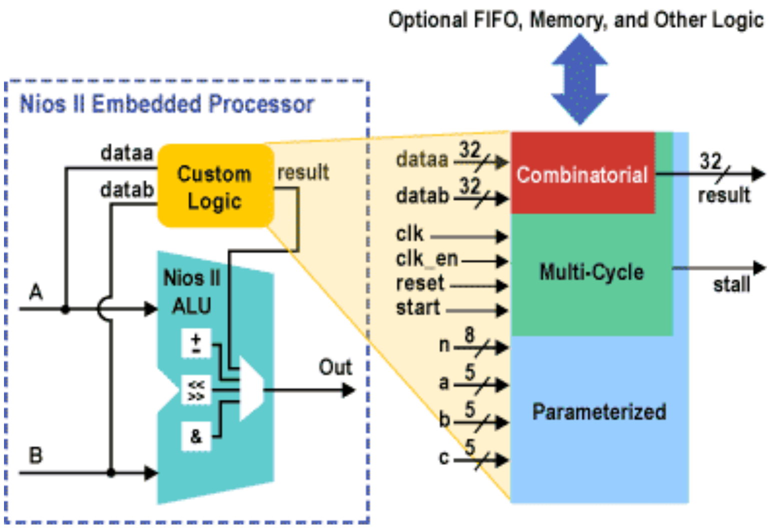
So far there's no support for floating-point arithmetic in NIOS II. Now you are going to add floating-point adder, subtraction and multiplier hardware, so that the elementary arithmetic operations behind the determinant can be performed faster, thereby reducing the operation execution time, in the expense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

By implementing a function in dedicated hardware you'll need to attach it to a bus in the processor, 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 block diagrams[1], Verilog or VHDL. 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**).

NIOS II CUSTOM INSTRUCTIONS

NIOS II supports different types of custom instructions depending on the type of hardware block connected to it.



http://www.altera.com/devices/processor/nios2/benefits/performance/ni2-acceleration.html

The supported custom instructions/logic blocks are:

- Combinatorial: single clock cycle logic block;
- Multi-cycle: fixed or variable number of clock cycles to complete execution;

[1] Recommended, but you can use any alternative.

[2] Again, recommended, but you can use another. There's a Modelsim tutorial by the end of the handout.

- Extended: block capable of performing multiple operations (e.g. ALU);
- 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 datapath.

For a complete description of custom instructions please use the material offered by Altera: http://www.altera.com/support/examples/nios2/exm-custom-instruction.html

FLOATING-POINT

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

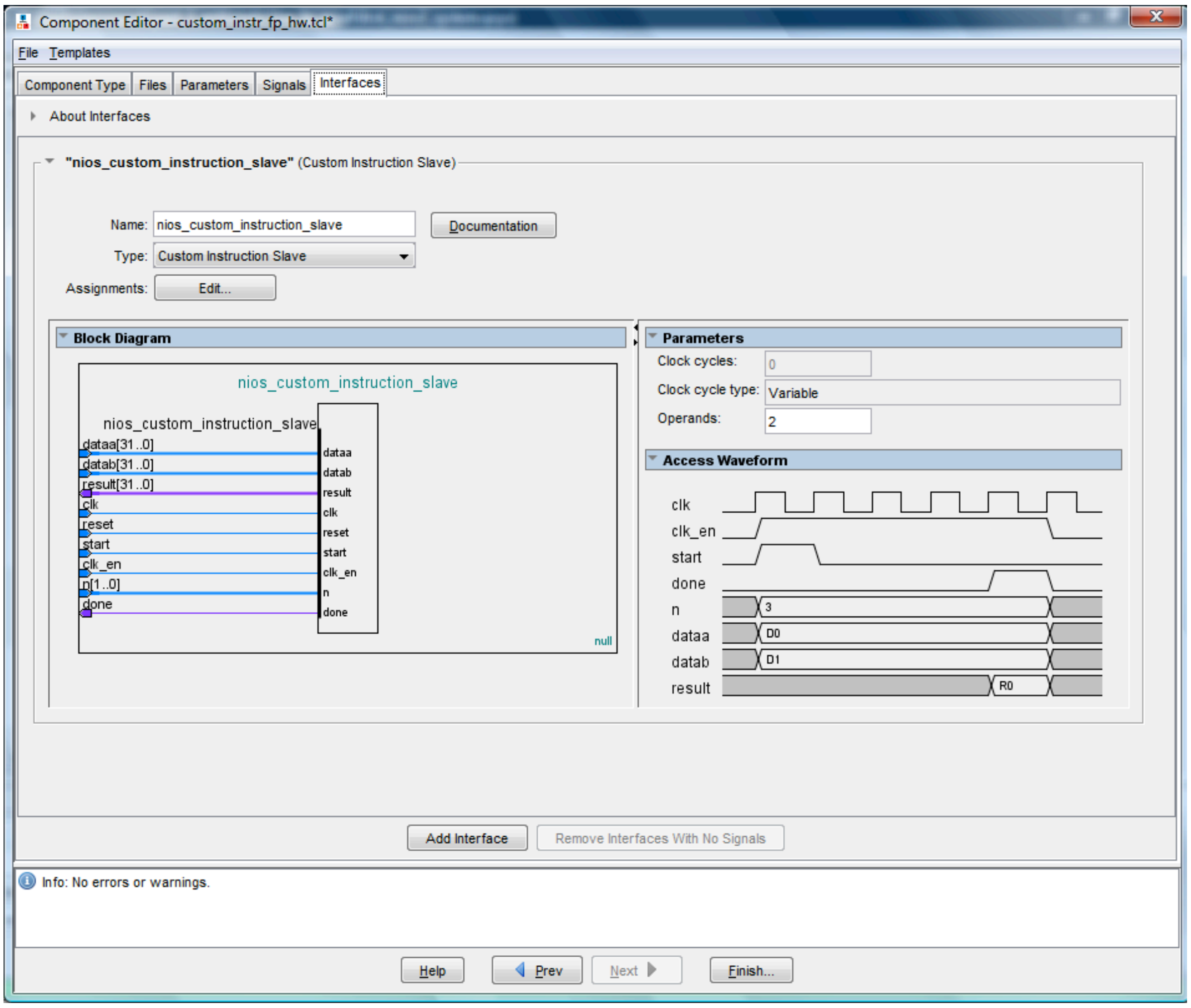
sign	exponent	significand
0	10000000	10010010000111111011011

Implement the custom logic blocks for floating-point arithmetic operations used for the evaluation of the determinant. Which type of custom logic blocks did you choose? Justify your choice. What conclusions do you make from this experiment in terms of circuit area *vs* code size *vs* execution speedup?

To implement custom instructions in a NIOS II-based system please read the following guide:

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

After having created your hardware design with the floating-point single precision adder, subtracter and multiplier, you can follow "NIOS II Custom Instruction User Guide" to complete your custom instruction.


Component Editor - custom_instr_fp_hw.tcl*
File Templates
Component Type Files Parameters Signals Interfaces
About Interfaces
Documentation
Edit...
Block Diagram
nios_custom_instruction_slave
dataa[31..0]
datab[31..0]
result[31..0]
clk
reset
start
clk_en
n[1..0]
done
null
Access Waveform
clk clk_en start done n dataa datab result
3 D0 D1 R0
Add Interface
Remove Interfaces With No Signals
Help Prev Next Finish...

"nios_custom_instruction_slave" (Custom Instruction Slave)

Name: nios_custom_instruction_slave
Type: Custom Instruction Slave
Assignments: Edit...

Parameters
Clock cycles: 0
Clock cycle type: Variable
Operands: 2

Info: No errors or warnings.

Hints:

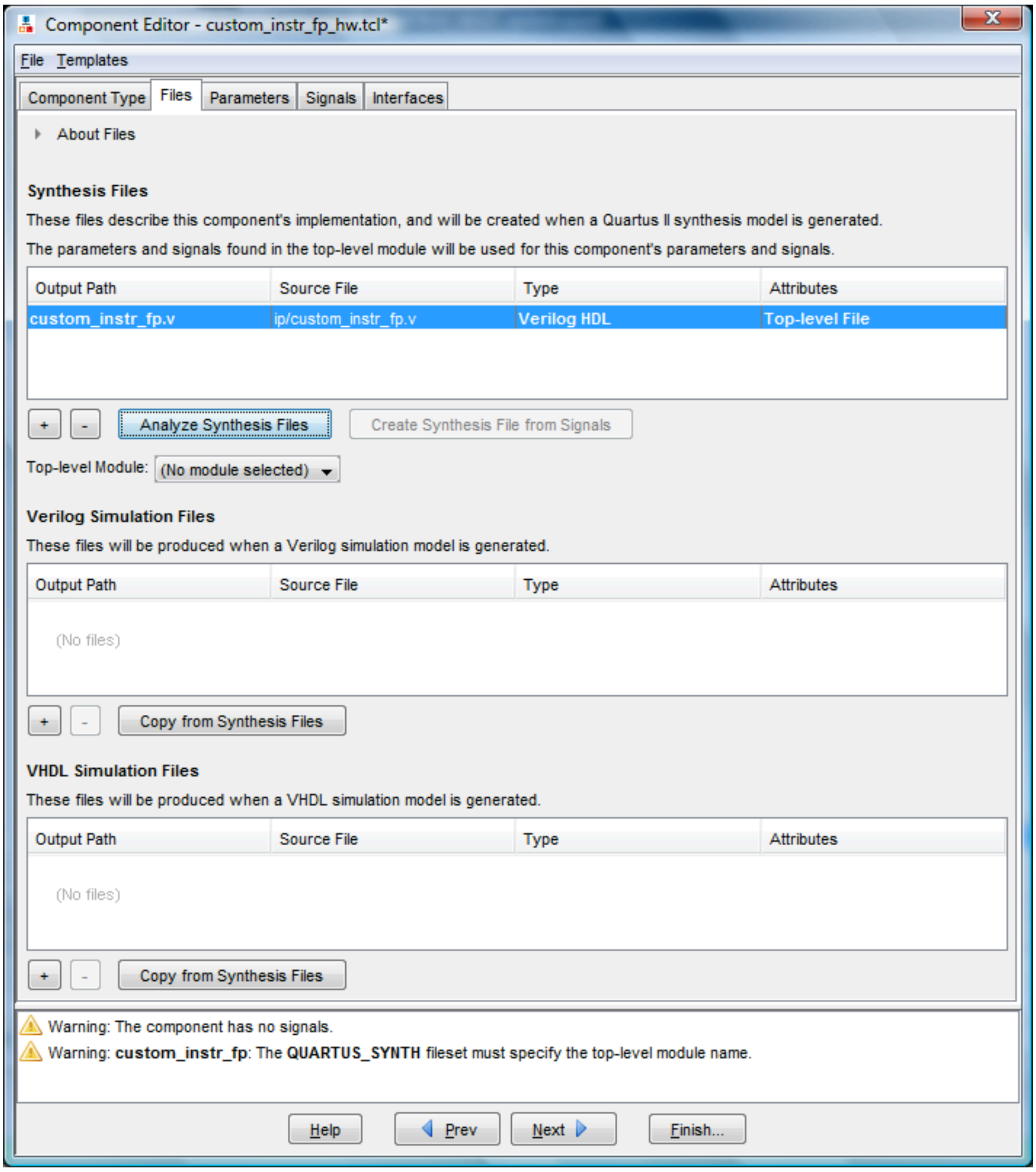
- Organize your hardware blocks in a subfolder named "ip" in your project folder;

- See http://www.altera.com/support/examples/nios2/exm-custom-instruction.html
- 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, it 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 VHDL 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;
- Keep in mind that Altera's CRC example is different than your floating-point add/sub/mult. You can't simply 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!**
- 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);`.

Debug:

- In case you don't read the expected value from your custom instruction, place a constant output of your hardware unit and verify if that value is read by the instruction.
- Whenever you introduce changes in your design, change the 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.

(Optional) Special numbers such as: 0, 1, NaN and ∞ are neutral, or absorbent, elements in addition and multiplication, according to the floating-point standard. **Bonus mark** (5%) if your custom hardware detects and produces results faster than having those values processed via the floating-point units by bypassing them.



Synthesis Files

These files describe this component's implementation, and will be created when a Quartus II synthesis model is generated.

The parameters and signals found in the top-level module will be used for this component's parameters and signals.

Output Path	Source File	Type	Attributes
custom_instr_fp.v	ip/custom_instr_fp.v	Verilog HDL	Top-level File

Top-level Module: (No module selected)

Verilog Simulation Files

These files will be produced when a Verilog simulation model is generated.

Output Path	Source File	Type	Attributes
(No files)			

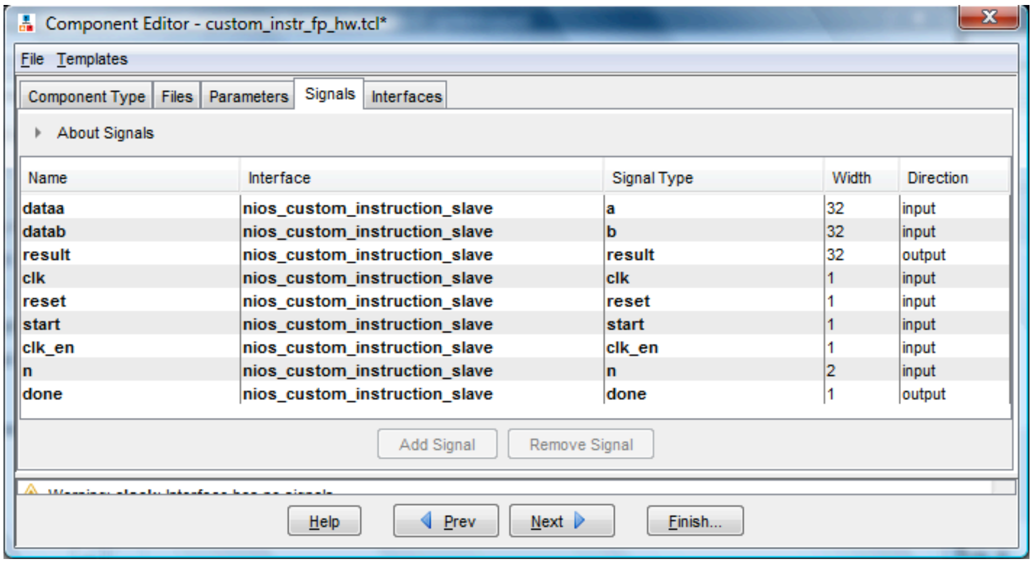
VHDL Simulation Files

These files will be produced when a VHDL simulation model is generated.

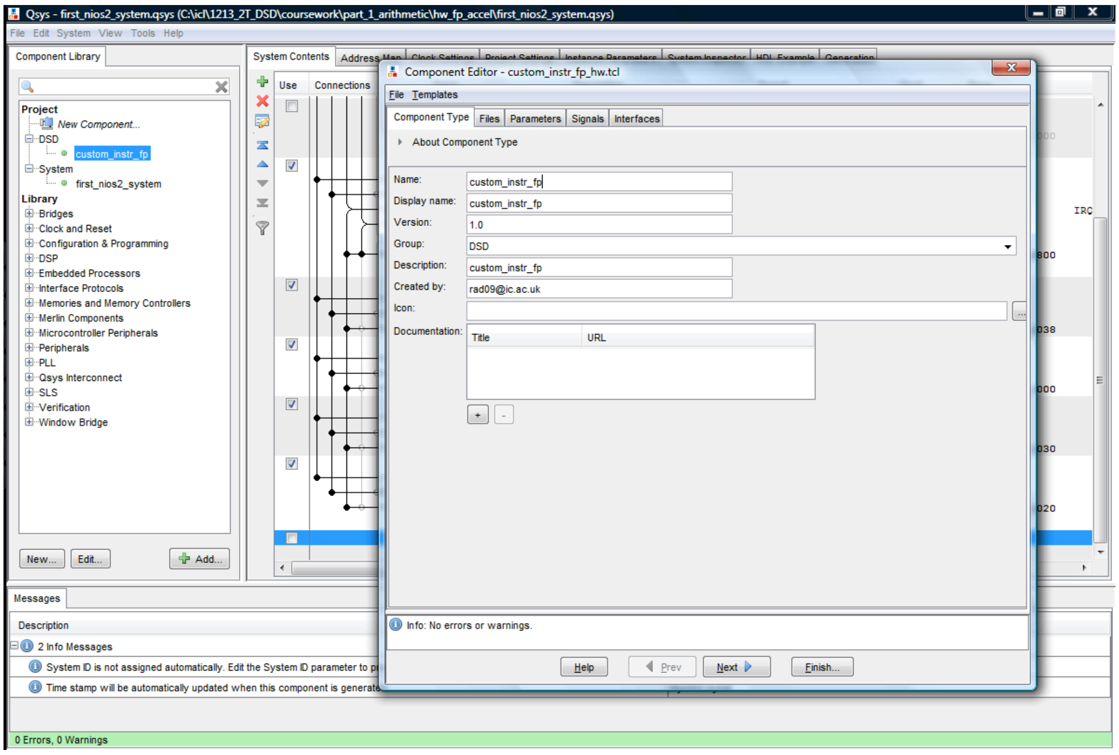
Output Path	Source File	Type	Attributes
(No files)			

Warning: The component has no signals.

Warning: **custom_instr_fp**: The **QUARTUS_SYNTH** fileset must specify the top-level module name.



Name	Interface	Signal Type	Width	Direction
dataa	nios_custom_instruction_slave	a	32	input
datab	nios_custom_instruction_slave	b	32	input
result	nios_custom_instruction_slave	result	32	output
clk	nios_custom_instruction_slave	clk	1	input
reset	nios_custom_instruction_slave	reset	1	input
start	nios_custom_instruction_slave	start	1	input
clk_en	nios_custom_instruction_slave	clk_en	1	input
n	nios_custom_instruction_slave	n	2	input
done	nios_custom_instruction_slave	done	1	output



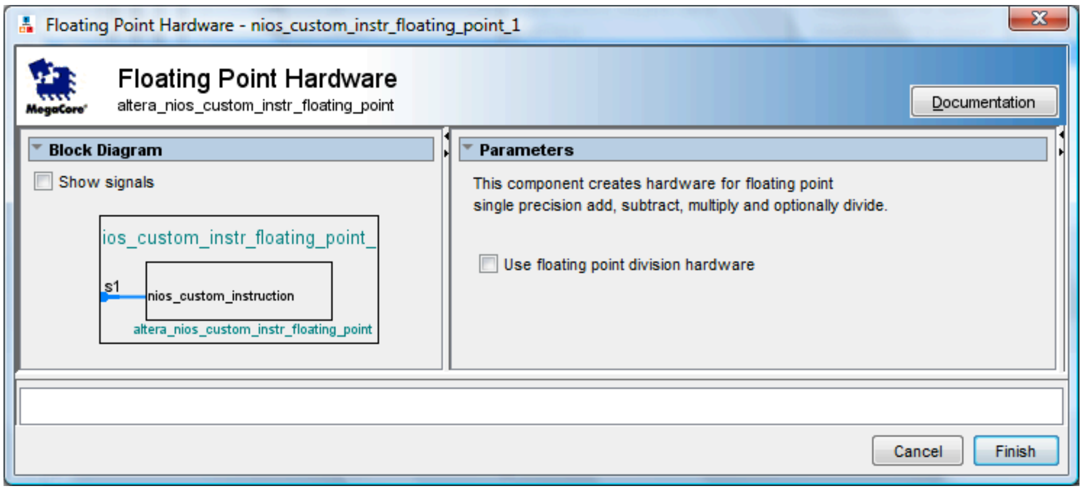
Name:	custom_instr_fp
Display name:	custom_instr_fp
Version:	1.0
Group:	DSD
Description:	custom_instr_fp
Created by:	rad09@ic.ac.uk
Icon:	
Documentation:	Title / URL

Info: No errors or warnings.

Description
2 Info Messages
System ID is not assigned automatically. Edit the System ID parameter to p
Time stamp will be automatically updated when this component is generate
0 Errors, 0 Warnings

Good to know:

In Qsys, you can add **Floating-Point Hardware** 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.


Floating Point Hardware - nios_custom_instr_floating_point_1
Floating Point Hardware
altera_nios_custom_instr_floating_point
Documentation
Block Diagram
Show signals
ios_custom_instr_floating_point_
s1
nios_custom_instruction
altera_nios_custom_instr_floating_point
Parameters
Cancel
Finish

This component creates hardware for floating point single precision add, subtract, multiply and optionally divide.

Use floating point division hardware

Dedicated Hardware Block to Compute the Determinant

You've seen that you can improve the performance of your system by implementing the most computationally demanding operations in hardware. In the present scenario you're going to implement a custom IP block that calculates the determinant of a matrix.

By executing some tasks in hardware there may be some benefit from certain operations for being done concurrently, as it will consume less time than the processor's iterative implementation. Depending on the system's configuration, you may even be able to execute different instructions on the processor while waiting for the hardware process to finish.

To reduce the processor time executing the determinant operation, you are going to create, in Quartus II, a dedicated hardware block to compute it. Later this hardware block will be connected to NIOS II using the Avalon bus.

Before implementing the design think about your design options:

- You can compute the determinant of large matrices at the expense of the determinant of their minors. Up to what dimension the determinant is to be calculated in hardware and in software? How do you make that decision?
- Data can be read from SDRAM memory in two ways: sequentially, or randomly upon request. The first option means that you will read all values at once and in sequence, like a DMA. In the second option the state machine of your dedicated hardware block will determine which data is passed to the block and when. This is done by sending requests to the memory controller to get specific memory locations. Justify your choice.
- Schedule your operations so that you don't have to wait until the last value to be read before you start producing intermediate results.

Hints:

- Place your hardware designs inside ./ip subfolder;

- Create and verify your hardware before adding it to the system. Pay attention to the latency of the floating-point units;
- You can connect your hardware to NIOS II via the Avalon bus. Look-up for the document "Avalon Interface Specification" http://www.altera.com/literature/manual/mnl_avalon_spec.pdf. If you want direct access to memory you need to see which ports you need to have in your hardware block to use it.
- Use the Component Editor to help you assigning your signals and interfaces. It will generate the waveforms for your interfaces!
- Be careful with the names of the hardware (IP) blocks in Qsys, they must be unique.
- As the complexity of the system scales, there are many things that can go wrong. Implement one feature at the time and verify that it works as expected.
- Plan ahead the control interface. That is to say, what are the functionalities you want your hardware block to have? E.g.: 1. Compute determinant; 2. Is it free to compute a new determinant? 3. Tell the hardware block where to get the data from, and how much; etc.
- On Altera's website and on www.alteraforum.com - there are some examples of more complex systems using the NIOS II processor. You're encouraged to see how they were done, and what are the trade-offs for each of them. You advised not to use them "as they are" as the increase on the complexity of your design and the problems that can arise from it will make you waste a lot of time.
- See Modular SGDMA and Avalon Memory Mapped Mater Templates:
    - http://www.altera.com/support/examples/nios2/
    - http://www.altera.com/support/examples/exm-list.jsp?cat=embedded
    - http://www.altera.com/support/examples/nios2/exm-avalon-mm.html
    - http://www.alteraforum.com/forum/showthread.php?t=19053

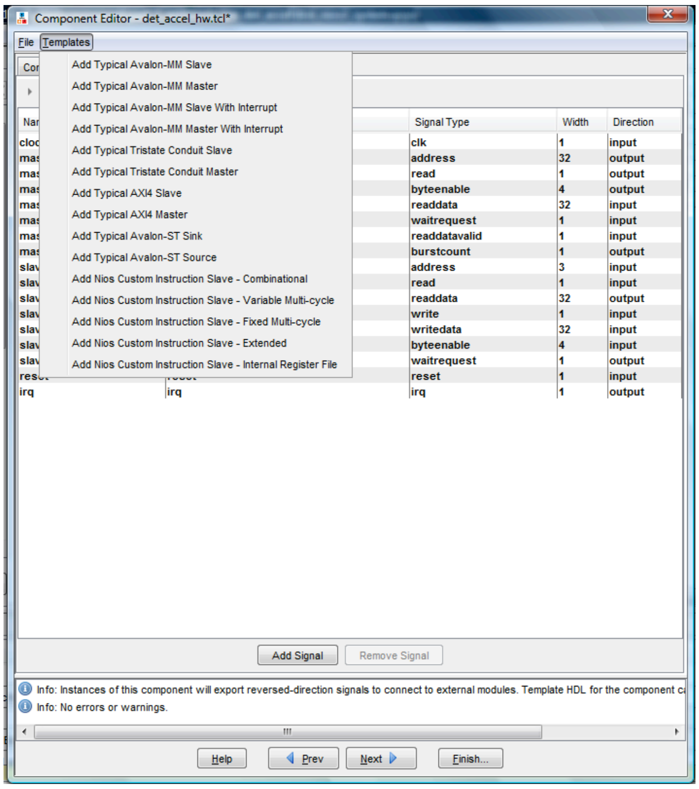
Once you have your system updated and running, answer the following questions:

- By how much did your system's performance improved? Show performance figures and compare them with the previous versions of your system.
- How much area does your hardware block require? And what about the size of the software? What conclusions can you draw in terms of circuit area/speed trade off?
- If you change the size of the CPU's cache, does it impact your results?
- Explain the changes you had to introduce in your system (software and hardware) to have the NIOSII CPU executing the determinant instructions on a custom hardware block. Don't forget to include your Qsys connection diagram.

(Optional) **Bonus mark** (3%) If you manage to have your processor executing other instructions while a determinant is being computed "at the same time".

Good to know:

When creating a new component you have the interfaces created from a template. Although you are not expected to use the generated VHDL/Verilog, you can use the (HDL output of) component editor to guide your schematic design. If you insert a new block in your system, you can use the Qsys to validate your interfaces! You can always replace the top design file, for the new block, with the one generated, by Quartus, from your block diagram.


Component Editor - det_accel_hw.tcl*
File
Templates
Cor
Add Typical Avalon-MM Slave
Add Typical Avalon-MM Master
Add Typical Avalon-MM Slave With Interrupt
Add Typical Avalon-MM Master With Interrupt
Add Typical Tristate Conduit Slave
Add Typical Tristate Conduit Master
Add Typical AXI4 Slave
Add Typical AXI4 Master
Add Typical Avalon-ST Sink
Add Typical Avalon-ST Source
Add Nios Custom Instruction Slave - Combinational
Add Nios Custom Instruction Slave - Variable Multi-cycle
Add Nios Custom Instruction Slave - Fixed Multi-cycle
Add Nios Custom Instruction Slave - Extended
Add Nios Custom Instruction Slave - Internal Register File
Add Signal
Remove Signal
Help
Prev
Next
Finish...

Nar		Signal Type	Width	Direction
cloc		clk	1	input
mas		address	32	output
mas		read	1	output
mas		byteenable	4	output
mas		readdata	32	input
mas		waitrequest	1	input
mas		readdatavalid	1	input
mas		burstcount	1	output
slav		address	3	input
slav		read	1	input
slav		readdata	32	output
slav		write	1	input
slav		writedata	32	input
slav		byteenable	4	input
slav		waitrequest	1	output
res		reset	1	input
irq	irq	irq	1	output

Info: Instances of this component will export reversed-direction signals to connect to external modules. Template HDL for the component c
Info: No errors or warnings.

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

```
Error (12006): Node instance "det_accel" instantiates undefined entity "first_nios2_system_det_accel"
```

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

`"first_nios2_system_det_accel"` with `"det_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.

PART II – STREAMING APPLICATIONS

In the second part of the coursework you are going to implement a dedicated hardware block to process a stream of data. Streaming applications are characterized by their (near) real-time requirements, or high throughput requirements. Usually the operations are simple and the same over time, but the overhead of instruction fetching and executing makes a generic processor like NIOS II unsuitable for such applications. Usually these applications involve a large number of streams to be processed, or the hardware area available in the system is reduced. Because of these two constraints, area and throughput, all arithmetic is done with integers.

The focus here is to design a custom IP block that applies an FIR filter to a stream of data. The targeted application is to remove a noise component from a music signal by applying a Notch Filter.

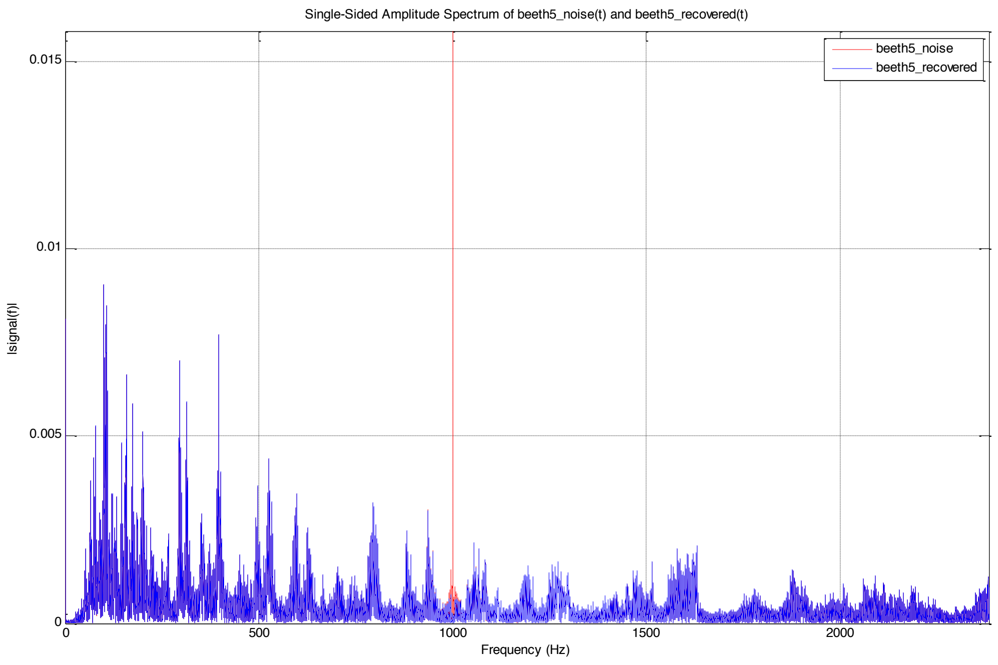
Notch Filter

The Notch filter is also known as band-reject filter or stop-band filter. It attenuates specific frequencies while leaving the others unaffected.

Along with the coursework material you will find a Matlab file containing a sound recording. If you load it and play it you'll hear part of a piece of music mixed with a continuous 1kHz tone. Your new system should be able to filter out the 1kHz tone, and you should be able to recover the original music in the end.

In the files provided for this coursework you will also find Matlab code which loads the sound and implements the Notch filter.

The figure below shows the FFTs for a sound (apx. the first 20 seconds of Beethoven's 5th Symphony), with the 1 kHz tone and the recovered signal after filtering with Matlab.



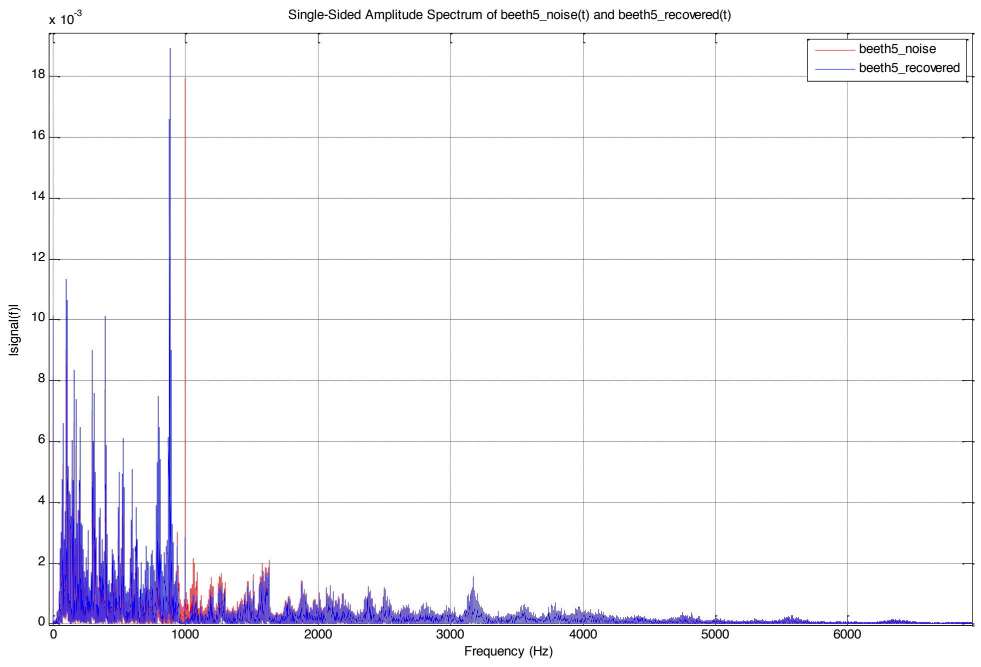
Your task is to design a system to filter in real-time the stream of data (music). To emulate the real-time system, while keeping complexity at a minimum you will read data from SDRAM and store the output in the SSRAM memory. Later, the contents of the SSRAM can be downloaded to the computer and processed in Matlab to demonstrate the effectiveness of your approach. All your arithmetic must be based on integer operators.

Decide which type of filter you are going to implement. FIR or IIR? Justify your decision.

Calculate the coefficients of your filter so that it removes the undesired frequency without filtering the remaining ones.

Hints:

- Pipeline your processing elements to increase their throughput.
- Quantize your filter coefficients in a way that you'll be able to speed-up your system, reduce the circuit area and minimize the distortion against the original signal.
- Apply Interval Arithmetic or Affine Arithmetic to calculate the number of bits of your signals and their ranges.
- The figure below shows the reconstruction of the signal with 8 bit truncation of the double-float filter coefficients. Not only it didn't remove the undesired frequency, as it filtered others that shouldn't be filtered.



- Determine the order of the filter you need to use, and make so that your hardware block takes the values for the coefficients from software. This way, you don't need to re-synthesize your design every time you need to change a coefficient.

- Use the Matlab script **mat2hex.m** to convert a Matlab variable with the sound into an Intel-hex file ready to be downloaded to the DE2-70 memories.
- Use DE2-70 control panel to download/upload data to/from the memories.

Hints:

- Have a look at this example to get *inspiration* on how to adapt your system to do an FIR: http://www.altera.com/support/examples/nios2/exm-accelerated-fir.html

MARKING SCHEME

Task	Mark
NIOS II w/ determinant software application	5 %
NIOS II w/ SDRAM	10 %
External Floating-Point hardware accelerator	15 %
External determinant hardware accelerator	30 %
Notch filter	40 %

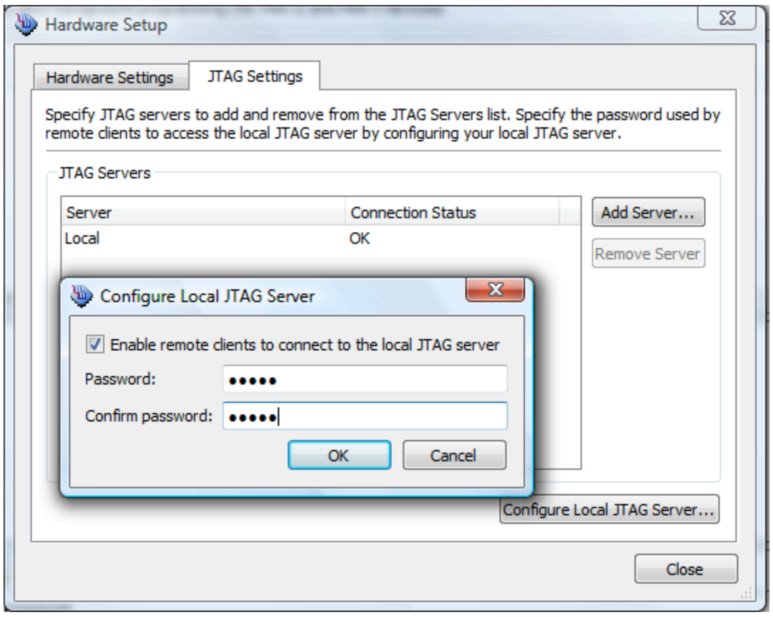
APPENDIX

DE2-70 Board

In the lab there are 5 boards, which can be programmed remotely. Due to limitations of the operating system, it may be required a user to start a session on the machine with the board attached, and then start the server via the programmer interface.

Steps to configure the server:

1. On Quartus II. Select: **Tools → Programmer**
2. Select Hardware Setup
3. JTAG Settings→Configure Local JTAG Server
4. Enable remote clients and set the password "ee506".



Specify JTAG servers to add and remove from the JTAG Servers list. Specify the password used by remote clients to access the local JTAG server by configuring your local JTAG server.

JTAG Servers

Server	Connection Status
Local	OK

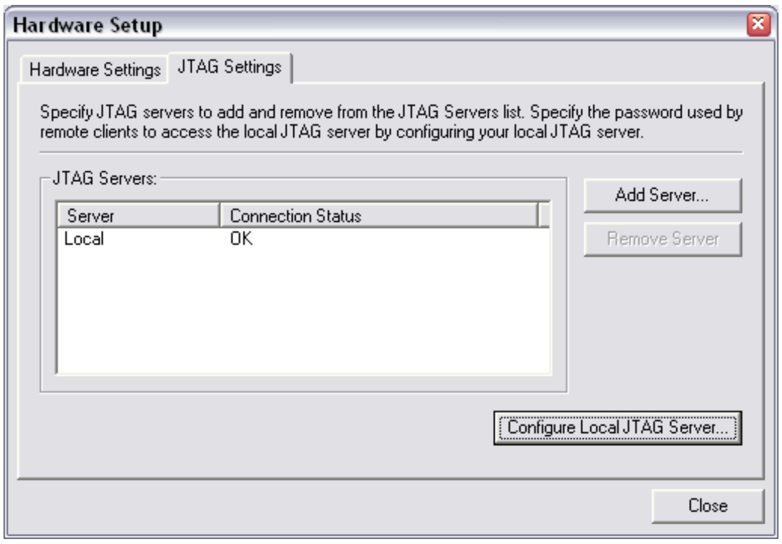
☑ Enable remote clients to connect to the local JTAG server

Password: •••••

Confirm password: •••••

The following steps describe how to program the DE2-70 board remotely.

5. On Quartus II. Select: **Tools → Programmer**
6. Select Hardware Setup (figure below)



Specify JTAG servers to add and remove from the JTAG Servers list. Specify the password used by remote clients to access the local JTAG server by configuring your local JTAG server.

JTAG Servers:

Server	Connection Status
Local	OK

7. Click on the JTAG settings tab and then on add server

8. Add the server address of a board (e.g. eews506a-013) and its password (e.g. ee506)

9. You should now see your board under JTAG servers (make sure the connection status is OK)

10. Under Hardware settings, select the board you have just added

11. Add your project and run!

12. Remember to always stop the NIOS II application in EDS, otherwise you may loose ability to reprogram the board and have to power cycle it before using it again.

List of boards in Level 5 lab:

- eews506a-013
- eews506a-033
- …

The password is always ee506

VHDL (Very) Quick Reference

Here you'll find information to quickly understand one of the hardware description languages, VHDL. It is used by the tool that runs functional simulations of your hardware. This is not meant to be a complete reference on VHDL. This is a quick reference so you can better understand the files that are generated by Quartus and simulated in Modelsim.

An example will be used to explain basic structure and syntax of VHDL. Think of it as the text version of block diagram, where you need to specify all the details about everything: blocks, interfaces, wires, etc.

```vhdl
-- example of a VHDL entity created by Quartus
-- this is a comment. All comments start with – and last until the end of the line
-- the VHDL file must have the same name as the entity. In this case: custom_instr_fp.vhd
-- VhDl Is NoT CaSe SeNsItIvE.

-- start by including the libraries that will be used (like C)
-- ieee.std_logic_1164.all covers most basic logic types
-- work lib is for your project. If you have more than one VHDL file
-- they'll all be in that library
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

-- definition of the entity (hardware block name), similar to a function prototype in C.
-- in the port section you define its interface: inputs and outputs
-- you can have a single wire "STD_LOGIC" or a bus "STD_LOGIC_VECTOR(31 DOWNTO 0)"
ENTITY custom_instr_fp IS
        PORT
        (
                clk :  IN  STD_LOGIC;
                reset :  IN  STD_LOGIC;
                start :  IN  STD_LOGIC;
                clk_en :  IN  STD_LOGIC;
                dataa :  IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
                datab :  IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
                n :  IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
                done :  OUT  STD_LOGIC;
                result :  OUT  STD_LOGIC_VECTOR(31 DOWNTO 0)
        );
END custom_instr_fp;

-- definition of the architecture – what other elements does it include
-- and how they are connected
-- you start specifying the name of the architecture and to what entity it belongs to
ARCHITECTURE bdf_type OF custom_instr_fp IS

-- definition or prototype of the components to be used in your architecture
-- each component has at least a name and definition of its ports
-- the 1st one only has 1 output of 5 bits and produces a constant output
COMPONENT lpm_constant0
        PORT(           result : OUT STD_LOGIC_VECTOR(4 DOWNTO 0)
        );
END COMPONENT;

-- more complex block than the first one. this is a 5 bit count-down counter.
-- This one has 4 inputs and 2 outputs.
COMPONENT counter
        PORT(sload : IN STD_LOGIC;
                 clock : IN STD_LOGIC;
                 clk_en : IN STD_LOGIC;
                 data : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
                 cout : OUT STD_LOGIC;
                 q : OUT STD_LOGIC_VECTOR(4 DOWNTO 0)
        );
END COMPONENT;


-- to make connections between blocks and interfaces you use "SIGNALS"
-- a signal can be a single wire "STD_LOGIC" or a bus "STD_LOGIC_VECTOR". Example: 5 bit bus
SIGNAL  clock :  STD_LOGIC;
SIGNAL  SYNTHESIZED_WIRE_5 :  STD_LOGIC_VECTOR(4 DOWNTO 0);
SIGNAL  DUMMY_WIRE_8 :  STD_LOGIC_VECTOR(4 DOWNTO 0);
SIGNAL  DUMMY_WIRE_9 :  STD_LOGIC;

-- up to this point you need to have specified all the units/sub-blocks and wires
-- that you are going to use
-- "begin" determines the start of description of your system
BEGIN

-- example of an (concurrent) assignment of a 5-bit constant value (binary by default)
-- all the "<=" assignments are done in parallel
DUMMY _WIRE_8 <= "00000";

-- assignment of the result of the logic AND (OR/XOR/etc.) between two signals
-- bit 0 of dummy_wire_8 and clk_end
-- assignments and operations MUST have to match word-length
```

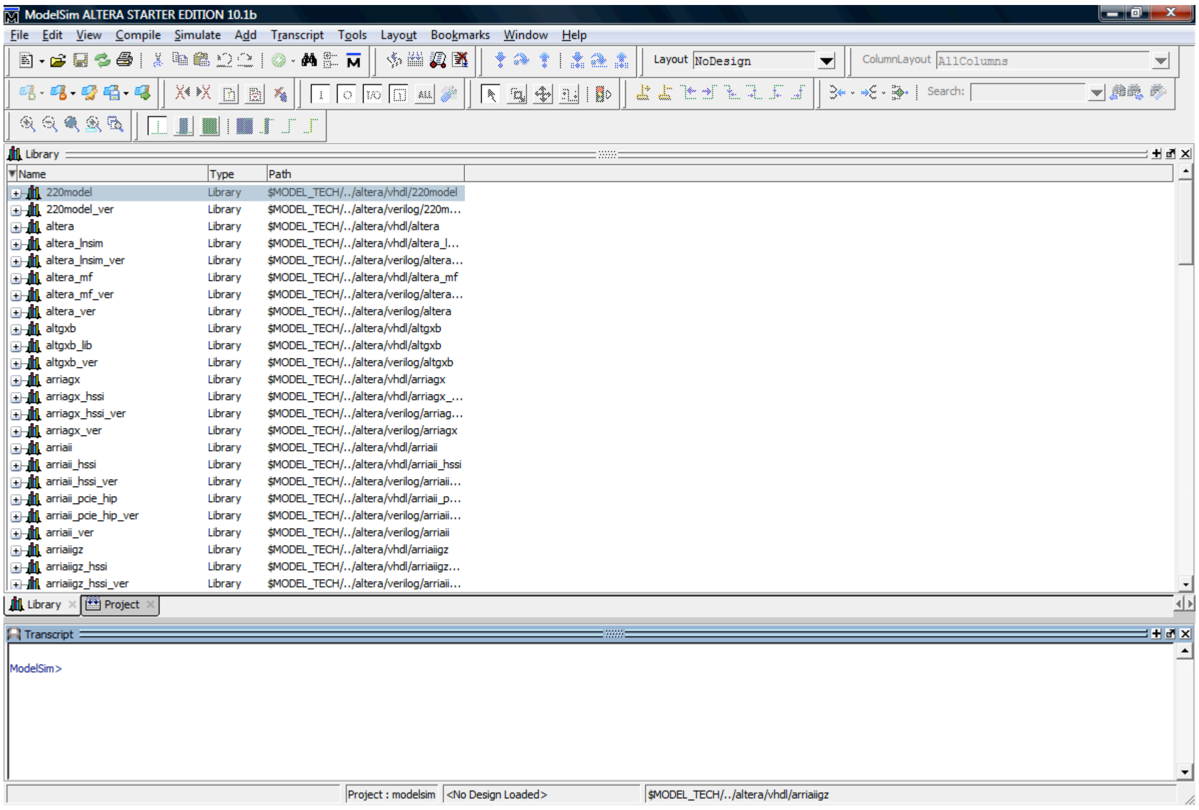
```
DUMMY_WIRE_9 <= DUMMY _WIRE_8(0) AND clk_en;

-- instantiation of the component "lpm_constant0" defined above.
-- you can have as many instantiations of the same component as you wish.
-- each instantiation must start with a different name (good to use meaningful names)
-- inside PORT MAP you specify the relation between the interface signals of
-- your component => your entity's architecture (local signal)
lpm_const_inst1 : lpm_constant0
PORT MAP(              result => SYNTHESIZED_WIRE_5);

-- instantiation of your other component.
-- similar to the previous one, but with more signals to connect
down_counter_inst : counter
PORT MAP(sload => start,
                clock => clock,
                clk_en => clk_en,
                data => SYNTHESIZED_WIRE_5,
                cout => done);

-- this is the end of your architecture's description
END bdf_type;
```

Modelsim Tutorial

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

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

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

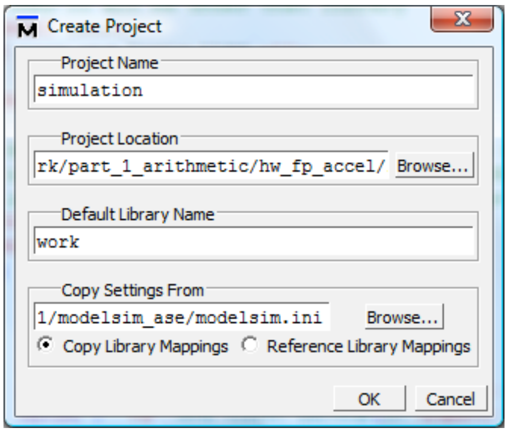
Depending on the task being done, Modelsim will change its interface to facilitate interaction or add extra functionality.

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/

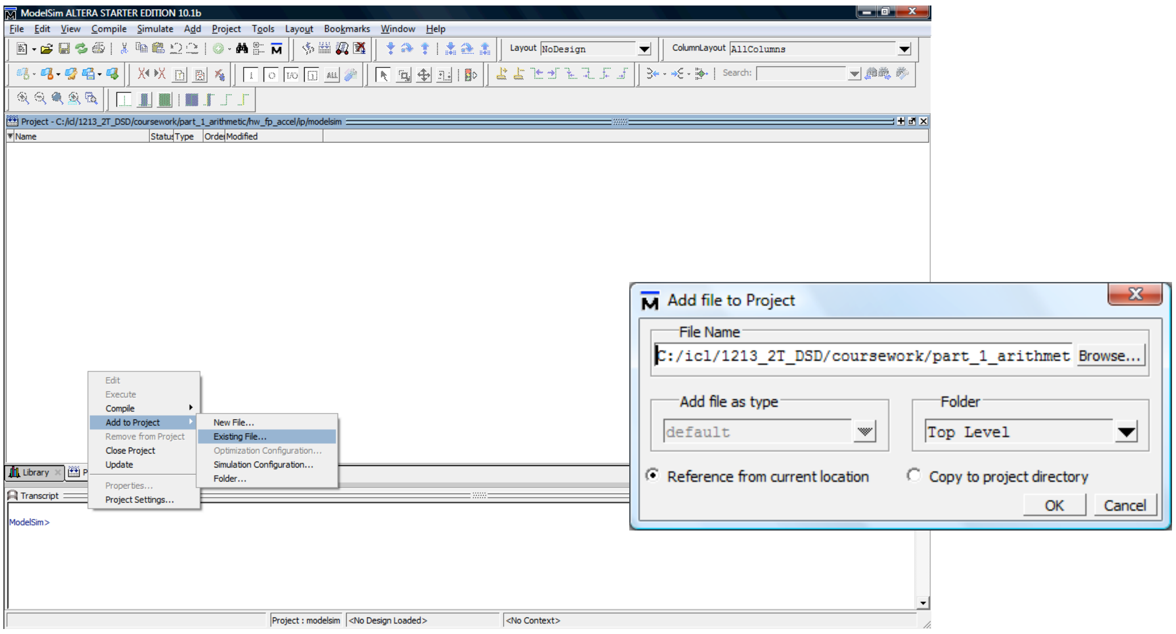
Default Library Name: work

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...

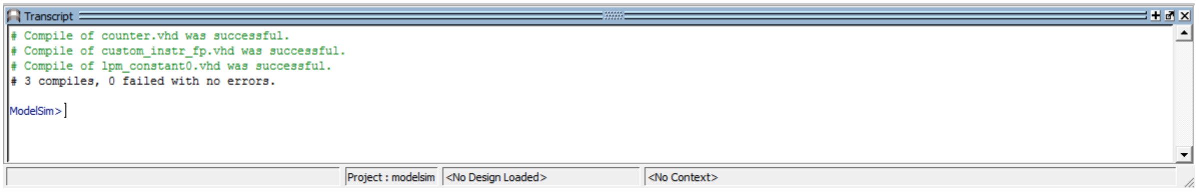

ModelSim ALTERA STARTER EDITION 10.1b
File Edit View Compile Simulate Add Project Tools Layout Bookmarks Window Help
Layout NoDesign
ColumnLayout AllColumns
Project - C:/icl/1213_2T_DSD/coursework/part_1_arithmetic/hw_fp_accel/ip/modelsim
Name Status Type Order Modified
Edit
Execute
Compile
Add to Project
Remove from Project
Close Project
Update
Properties...
Project Settings...
New File...
Existing File...
Optimization Configuration...
Simulation Configuration...
Folder...
Library
Transcript
ModelSim>
Project : modelsim
<No Design Loaded>
<No Context>
Add file to Project
File Name
C:/icl/1213_2T_DSD/coursework/part_1_arithmet
Browse...
Add file as type
default
Folder
Top Level
Reference from current location
Copy to project directory
OK
Cancel

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> ]
```

SIMULATION

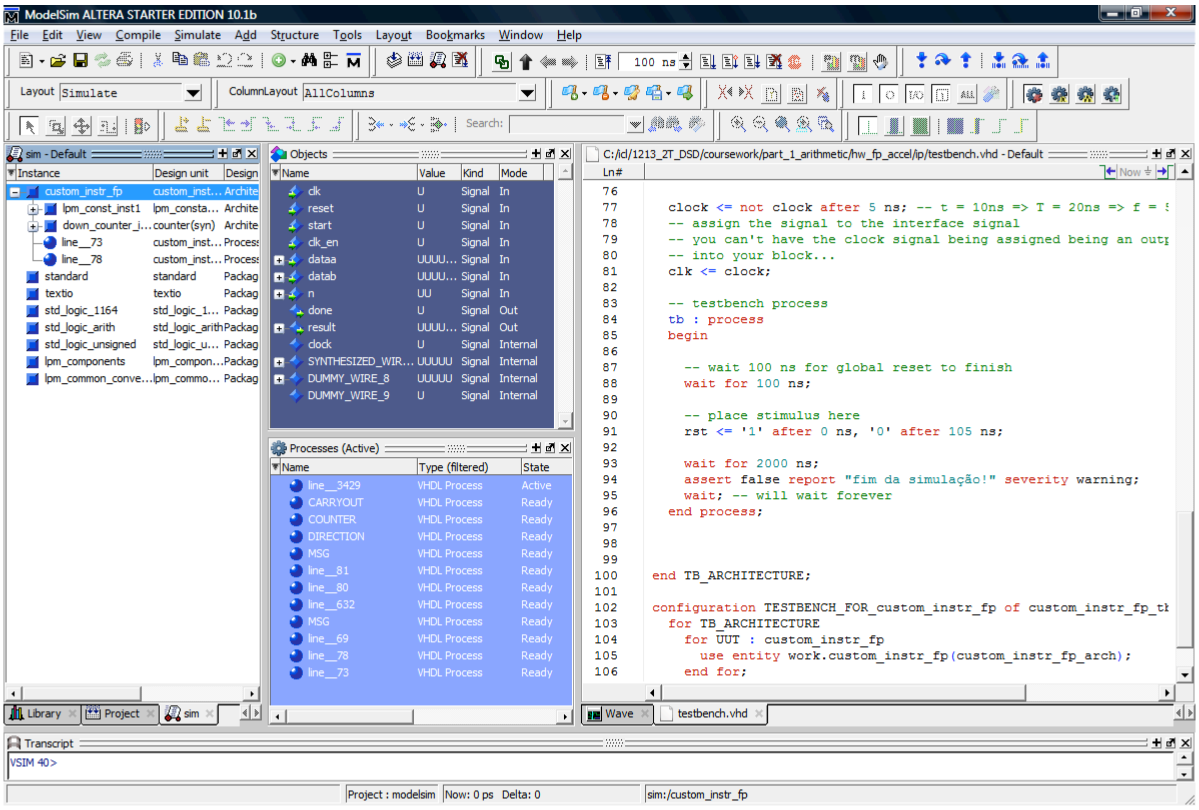
To run a simulation you need to specify a stimulus for your design, so you can check it works accordingly.

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

Layout: Simulate ColumnLayout: AllColumns

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

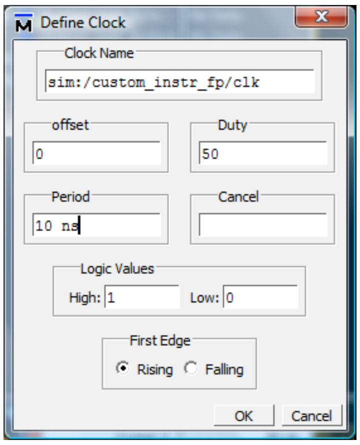
```
	clock <= not clock after 5 ns; -- t = 10ns => T = 20ns => f = 5
	-- assign the signal to the interface signal
	-- you can't have the clock signal being assigned being an outp
	-- into your block...
	clk <= clock;

	-- 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


Define Clock

Clock Name: sim:/custom_instr_fp/clk

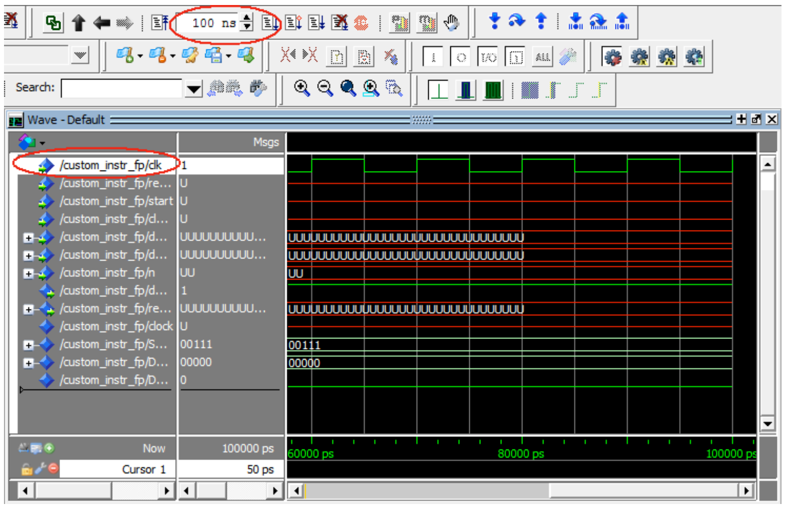
offset: 0 Duty: 50

Period: 10 ns Cancel:

Logic Values — High: 1 Low: 0

First Edge: [✓ Rising] Falling

OK Cancel

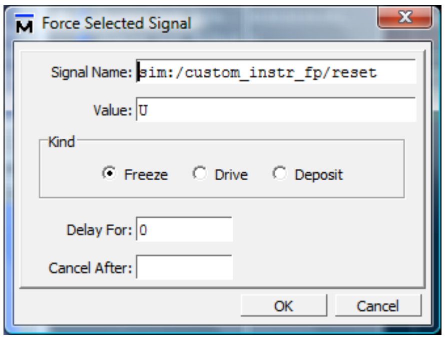

100 ns

Wave - Default

	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…	UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
/custom_instr_fp/d…	UUUUUUUUUU…	UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
/custom_instr_fp/n	UU	UU
/custom_instr_fp/d…	1	
/custom_instr_fp/re…	UUUUUUUUUU…	UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
/custom_instr_fp/clock	U	
/custom_instr_fp/S…	00111	00111
/custom_instr_fp/D…	00000	00000
/custom_instr_fp/D…	0	

Now 100000 ps Cursor 1 50 ps 60000 ps 80000 ps 100000 ps

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


Force Selected Signal

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 an 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 and reset signals.

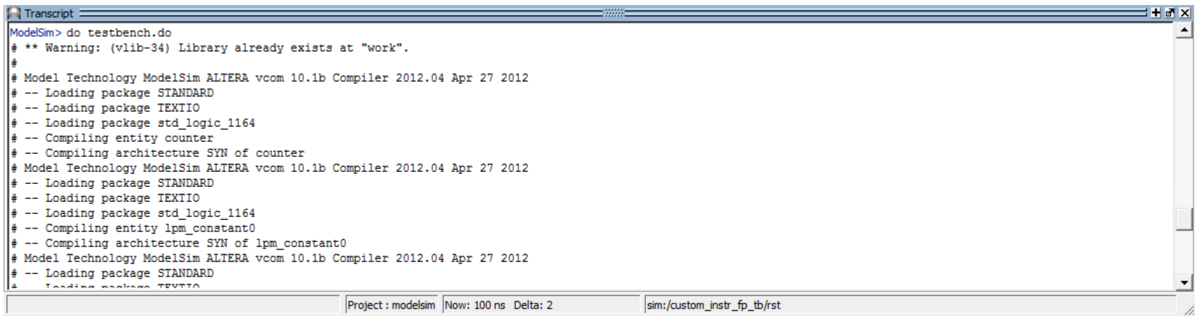
To avoid having to define the wave diagram for every simulation, it is possible to automate the process running a macro: testbench.do

```
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

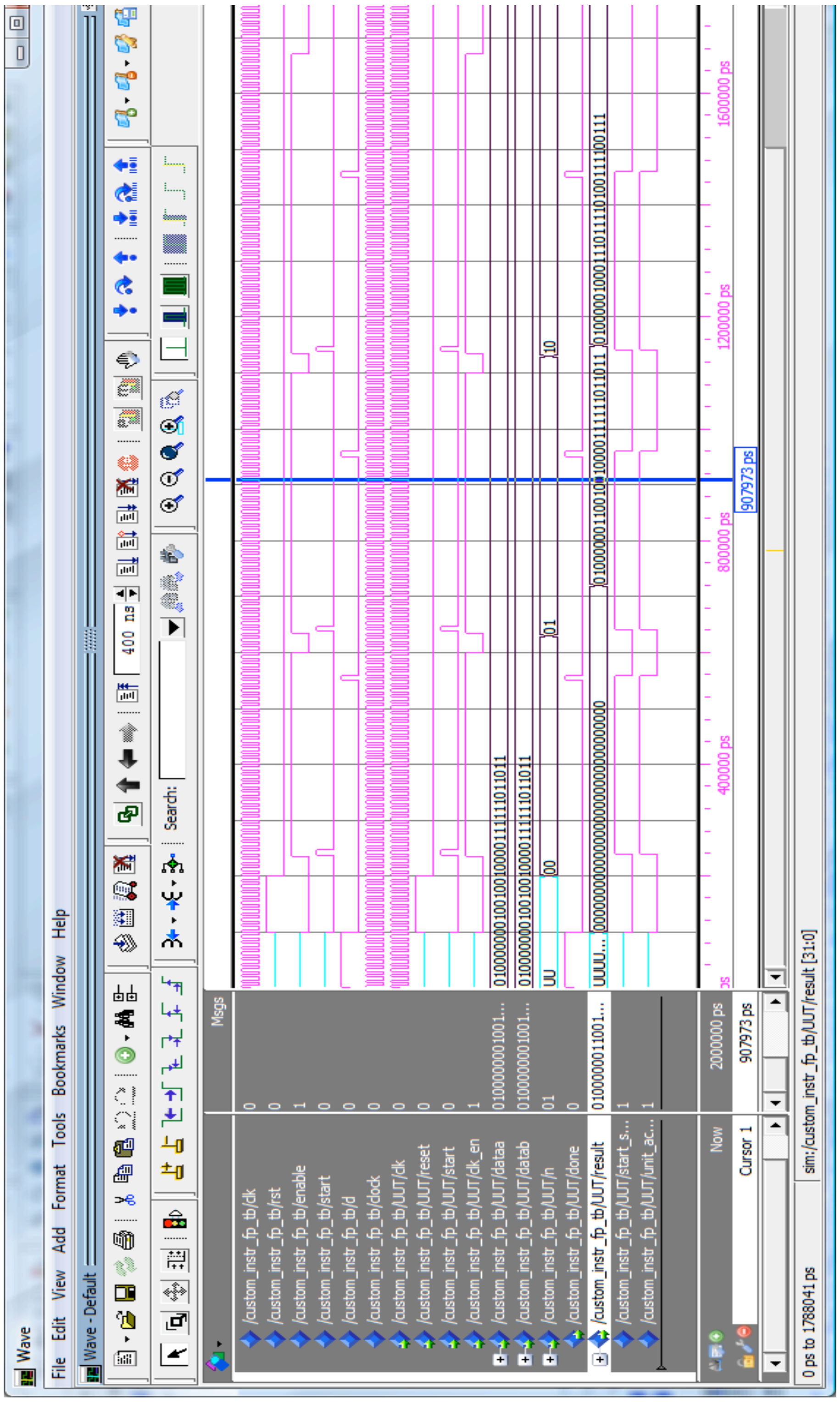
These files cover the basic simulation functionalities and you can adapt them to simulate your hardware blocks.



```
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
```

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.


Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Search:
Msgs
/custom_instr_fp_tb/clk
/custom_instr_fp_tb/rst
/custom_instr_fp_tb/enable
/custom_instr_fp_tb/start
/custom_instr_fp_tb/d
/custom_instr_fp_tb/clock
/custom_instr_fp_tb/UUT/clk
/custom_instr_fp_tb/UUT/reset
/custom_instr_fp_tb/UUT/start
/custom_instr_fp_tb/UUT/clk_en
/custom_instr_fp_tb/UUT/dataa
/custom_instr_fp_tb/UUT/datab
/custom_instr_fp_tb/UUT/n
/custom_instr_fp_tb/UUT/done
/custom_instr_fp_tb/UUT/result
/custom_instr_fp_tb/UUT/start_s...
/custom_instr_fp_tb/UUT/unit_ac...
Now
Cursor 1
2000000 ps
907973 ps
400000 ps
800000 ps
1200000 ps
1600000 ps
0 ps to 1788041 ps
sim:/custom_instr_fp_tb/UUT/result [31:0]

Application Profiling

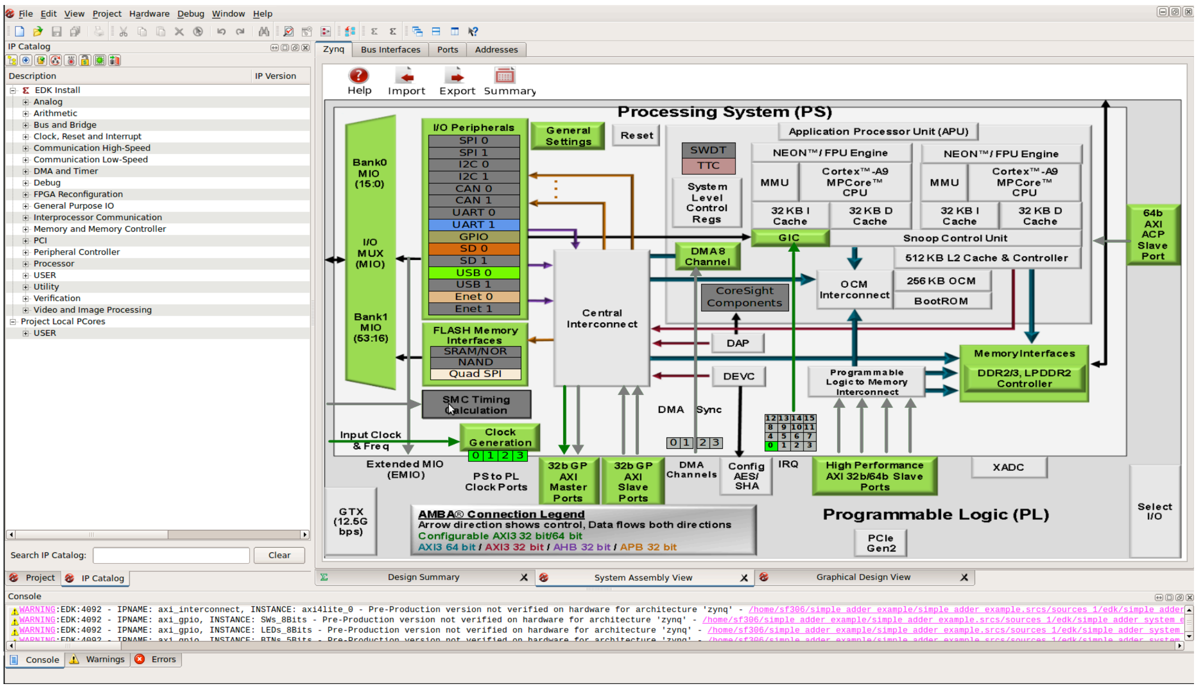
In the quest to accelerate complex applications, it may not be obvious which instructions take most of the processor’s time. To measure the performance of parts of your application you can make use of an external profiler or measure the time require. Altera provides an application note specific on this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

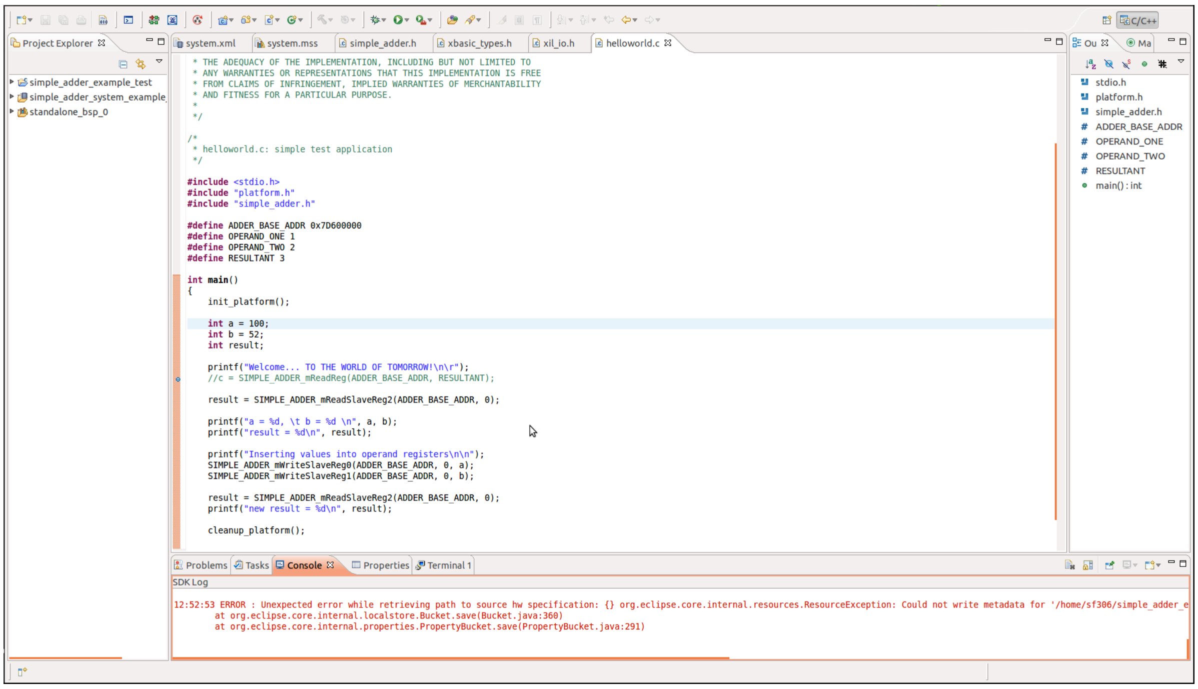
Here are screenshots of the development environments for hardware (XPS) and software (SDK). You'll find them similar to the tools you're using in this coursework.

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 ***