Lab Assignments in TDT4255 Computer Design

NTNU

Computer Architecture and Design Group
Department of Computer and Information Science

Version 2: 30th of April 2012

Contents

List of Figures

Abbreviations

ALU Arithmetic Logic Unit

ASIC Application Specific Integrated Circuit

BRAM Block RAM

BSB Base System Builder

CIP Create and Import Peripheral

CLB Configurable Logic Block

DUT Design Under Test

FF flip–flop

FPGA Field Programmable Gate Array

FSM Finite State Machine

GP General Processor

HDL Hardware Description Language

IC Integrated Circuit

I/O Input/Output

IP Internet Protocol

LUT Look–up Table

MUX Multiplexor

PLB Processor Local Bus

RAM Random–Access Memory

RISC Reduced Instruction Set Computer

RTL Register Transfer Level

Si silicon

SRAM static Random–Access Memory (RAM)

UART Universal Asynchronous Receiver/Transmitter

USB Universal Serial Bus

UUT Unit Under Test

VHDL VHSIC HDL

VHSIC Very High Speed Integrated Circuit

XPS Xilinx Platform Studio

Chapter 1

Introduction

This compendium is an accompaniment for the set of lab assignments in the course TDT4255 Computer Design which is given by the Computer Architecture and Design group. It contains the description of the lab assignments, the description of hardware and tools to be used and some practical information. Because the tools are rather complex, the whole Chapter 2 is devoted to the introduction of the tools and development environment which will be used for assignments. Each of the three chapters which follow contains a description and clarification for one of the course assignments.

Lab assignments are graded and these grades are part of the final grade in the course. Therefore, it is to your best interest to carefully read this compendium and understand its contents.

1.1 Practical Goal: the Processor Architecture and Design

The main goal of the assignments is the design and implementation of a central part of each computer – the processor. You will do this based on the knowledge of computer architecture and computer hardware design which you will acquire through the course lectures. The processor will be implemented on an FPGA chip from the Spartan 6 family by Xilinx. Spartan 6 chip is placed on the development board by Avnet with additional hardware resources which make it possible to test the processor within a larger system.

You will implement different processor architectures i.e. multicycle and pipelined architectures thereby obtaining practical knowledge about the operation of each, their advantages and drawbacks. Assignments are presented in a way which will give you a logical learning path for the processor architecture from ALU to the implementation of the processor with its control and data paths.

1.2 Learning Outcome

The main learning outcome is:

- **the knowledge of the processor core architecture**

In addition, the lab assignments are organised in such a way which will provide you with practical knowledge of computer hardware design, particular steps of the design and implementation processes, reconfigurable chips, use of VHDL, embedded systems design and use of advanced development environments such as Xilinx ISE Design Suite, in particular ISE Project Navigator and Xilinx Platform Studio, XPS.

In brief, you will get the experience with the following:

- Hardware design in VHDL
- Steps of hardware design within a complex development environments such as Xilinx ISE
- Design simulations in ModelSim
- Designing and programming for embedded systems (XPS)
- FPGAs

1.2.1 A Brief Overview of Hardware and Tools

The first hours of practical work in the lab are intended for familiarisation with hardware and tools you will be using for the lab assignments. Therefore, we have made a brief tutorial which makes the most of the Chapter 2 contents. In order to introduce you to the sort of assignments which await you in this course, you will complete a simple task through this tutorial.

1.2.2 Assignment 1

You will design and implement a simple multi-cycle MIPS processor in VHDL and synthesise your design.

1.2.3 Assignment 2

In Assignment 2, you will design and implement a pipelined processor architecture.

1.2.4 Assignment 3

Assignment 3, you will extend your previously implemented pipelined processor to optimize its performance by implementing different hazard detection and correction techniques.

1.3 Practical Information

Some practical information is provided in order to ease the process of preparing and delivering assignment results but also to prevent misunderstandings regarding the content and grading of your deliveries.

1.3.1 Lab and Assistance

For this course you will be working in groups of two. You are free to choose your group partner. In case you cannot find a fellow student to work with, contact a teaching assistant for the course. He will be able to find a lab partner if there are more students missing one.

The lab premises at which you will be working are on the fourth floor of the IT-west building, room 458.

1.3.2 Deliveries

A delivery for each assignment should contain the following items:

- Report
- VHDL files with the design
- VHDL files with the test benches
- Source code of the test programs for the implemented processors

Remember to comment your VHDL code.

Report

A report is the most important part of the delivery. It not only presents your work, also it shows how well you have understood the task and acquired the needed knowledge. Therefore, it is important to spend some time studying the tips on how to write a good report before you begin with writing one.

Firstly, a good report does not have to be a long one. On the contrary, reporting is all about concise communication of the main ideas and solutions regarding the report subject. Of course, the number of pages depends on the concrete assignment and on the extent of your solution so it will vary according to the need for a thorough description of your work. However, for the set of assignments in this course, an average of 10 pages would suffice.

The style of writing need be particularly stripped off of all unnecessary information. The sentences should be clear, presenting precisely the idea you wish to convey. Only the facts which are needed for providing a good picture of your work should be kept.

Whenever you can present your results or ideas in figures or tables, do that! One picture is worth thousands words. Of course, a figure or a table needs to be thought up well so that it conveys the needed information in the concise and easily understandable way. Then, remember to make references to figures and tables throughout the text.

Moreover, references should be made to the sources of information such as books, datasheets and similar, which you consult for writing a report. It is a sign of a good writing style for a formal document.

A report should be organised hierarchically. While you are free to choose the exact organisation, you should keep it within generally accepted framework for report organisation. According to this, a report should contain following basic sections:

Abstract – contains an overview of the work on the assignment. It provides a brief description of the task and the achievements and results of the work presented in the report. If such is the case, it also mentions the things which have not been successfully implemented.

Introduction – introduces the task of the assignment and the challenges it brings. Also, it gives a brief introduction to how the task was approached and in which way the solution was reached.

Solution – describes your solution of the task. Contains a detailed description of all the subtasks which have been solved and how they contribute to the solution for the given task. The use of diagrams, figures, tables and similar is welcome as a support to your description.

Result – presents the results: what has been successfully completed and what did not work. If any ways around it were found, provide them at this place. Every solution should be tested for its validity. This is the place where you will describe what kind of testing you have performed and what the outcome of your tests was.

Discussion – Discuss the assignment and your achievements. You are free to critically assess your work – what could have been done better, which way you would choose to go if given the same task again etc.

Conclusion – a brief conclusion of the performed work. Round–up the challenges and results

Bibliography – follows a report as a list of references which have been used in the report.

1.3.3 Evaluation

Assignment deliveries are evaluated based on the delivered report and code. The number of points you will score for the assignment is decided upon the following:

- To what extent the requirements of the assignment have been fulfilled
- The quality of the delivered report
- Code quality and technical solutions
- Testing
- Solutions which go beyond the assignment requirements

Chapter 2

A Brief Overview of Hardware and Tools

The goal of this chapter is to introduce you to the hardware and accompanying tools you will be working with on the course assignments. The content is kept as simple as possible. However, the tools you will be working with are rather complex so if you would like to look for more information about specific features, a number of references to appropriate documents are provided throughout the text. Moreover, as *learning by doing* has proved to be an efficient way of grasping new knowledge, we have provided a brief tutorial to familiarise you with the hardware and tools you are going to use throughout the semester.

2.1 Introduction

All three assignments are about computer design and implementation in hardware. You will be asked to design a computer unit i.e. ALU unit or processor core and implement the design in a chip. For the design, a Hardware Description Language (HDL) will be used. In particular, you will work with VHSIC HDL (VHDL) within a Xilinx ISE development environment [4]. You will implement your design in a reconfigurable chip, a Field Programmable Gate Array, FPGA chip. In particular, you will work with a chip from the Spartan 6 family by Xilinx which will be used within an S6LX16 development board by Avnet. The board contains a number of other units which enable the access to Spartan 6 chip for its configuration as well as testing during its operation. Figure 2.1 shows the development board S6LX16.

Figure 2.2 shows a schematic view of the hardware with which you will work. Different tools will be used for different stages of the development of the solution. The snapshots of the tools are also shown in Figure 2.2 in relation to the parts of the hardware setup they are used for.

2.2 VHDL

A brief overview of the main VHDL features follows.

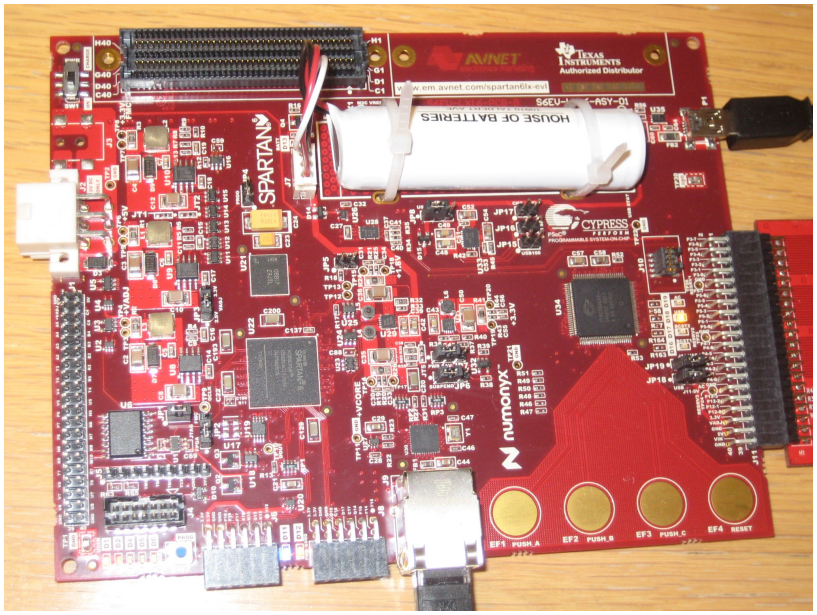

Figure 2.1: Avnet S6LX16 Development Board

2.2.1 Introduction

VHDL is an extensive language. We encourage all who need more information to buy a book: 'The VHDL Cookbook' [1]. It is old but it describes the VHDL syntax very well. However, it is not so good to describe what it is that distinguishes VHDL from programming languages and how VHDL should be written so that the generated hardware is synthesisable into functional units.

VHDL is a language intended for specification of digital circuits. Originally, the intention was to provide a brief and clear documentation of the circuits but soon more possibilities were discovered. In the first place, it was the possibility for the simulation of the VHDL code with the aim of checking that the circuits perform as they should. After some time, it became possible to synthesise the circuit description in VHDL. This meant an automatic conversion of the VHDL code into actual logic circuits, either in Field Programmable Gate Array (FPGA) or Application Specific Integrated Circuit (ASIC). Then it became possible to make a complete design of digital circuits through a description in VHDL and then let the tools generate implementation files for FPGA or ASIC production.

Although the whole VHDL language can be simulated by a VHDL simulator, only a subset of VHDL can be synthesised. This often brings in problems for fresh VHDL designers who write the code which can be nicely simulated but cannot be synthesised. Therefore, it is useful to have synthesis in the back of the mind during the design process so that no unpleasant surprises pop up when the circuit comes to the synthesis stage. For good tips on how to write a synthesisable VHDL code, we recommend 'HDL Coding Techniques' chapter in XST manual [2].

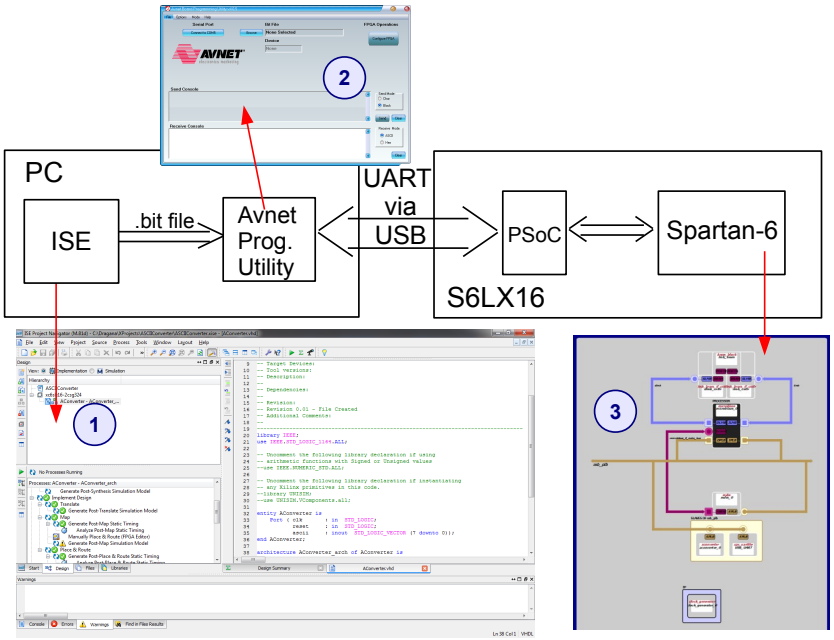

Figure 2.2: Hardware setup and accompanying tools: 1 – Xilinx ISE Project Navigator within which the configuration bitstream and a corresponding .bit file are generated; 2 – Avnet Programming Utility which transfers .bit file to the S6LX16 board via UART over USB connection; 3 – embedded system implemented in Spartan 6 (block diagram generated within XPS)

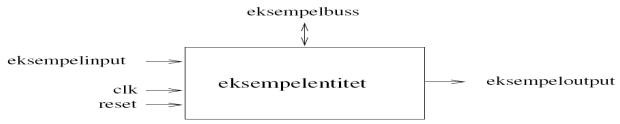

Figure 2.3: Example entity

What is important for writing a synthesisable code is to think in the right way during the design phase. You are all familiar with programming and VHDL looks like a programming language. However, it is only so at the first glance. When you are making circuit modules, you need to think as a digital designer. Those who do not do so, often end up with writing a bad code or the code which is impossible to synthesise. You have to see for yourselves about flip–flops (FFs), Multiplexors (MUXs), buses and combinatorial logic. This level of design is called Register Transfer Level (RTL). To come to that stage, you need some experience.

2.2.2 Structure

Typically, a simple VHDL file implements a simple hardware module and consists of three parts. The first part states which libraries will be used. This is something which corresponds to the inclusion of header files in C. An example which includes the library 'ieee' and which specifically uses the package 'std_logic_1164' from this library would look like this:

```
library ieee;
use ieee.std_logic_1164.all;
```

The second part consists of entity description. An entity shows how a hardware module communicates with its environment, which signals go in and out of the module. It is something which corresponds to the interface in Java. Here is an example which defines a hardware module with three input signals, one output signal and one 8–bit bidirectional bus:

```
entity eksempelentitet is

 port (
   eksempelinput        : in    std_logic;
   eksempelbuss         : inout std_logic_vector(7 downto 0);
   eksempeloutput       : out   std_logic;
   clk                  : in    std_logic;
   reset                : in    std_logic);

end eksempelentitet;
```

This entity corresponds to the circuit depicted in Figure 2.3.

The last part of a VHDL file contains the implementation of the corresponding entity. This is called architecture and it is the place where the logic is

specified. One example of the architecture would be as following:

```
architecture eksempelarch of eksempelentitet is

begin

        -- this is a comment
        -- here comes the implementation itself

end eksempelarch;
```

The architecture can instantiate modules defined in other VHDL files and, therefore, a hierarchy can be made of VHDL modules which together make up a complex system. Here is an example of the architecture which instantiates our example module:

```
architecture fu of bar is

  signal eksempelinput_i : std_logic;
  signal eksempelbuss_i  : std_logic_vector(7 downto 0);
  signal eksempeloutput_i: std_logic;

begin

  eksempelmodul: eksempelentitet
    port map (
      eksempelinput     => eksempelinput_i,
      eksempelbuss      => eksempelbuss_i,
      eksempeloutput    => eksempeloutput_i,
      clk               => clk,
      reset             => reset);

end fu;
```

Pay attention to the fact that the architecture defines three new internal signals, just above `begin`. Simply stated, a signal is a conductor which is, among other things, used as a connection between modules within the system. In our example, you can see how each signal in the example module is mapped to one of the internal signals in the architecture `fu of bar` which instantiates the module. Therefore, it is natural that these internal signals are further used at some other place within this architecture, either to connect the example module with some other instantiated module or for the logic specified in this architecture (architecture `fu of bar`).

2.2.3 The Description of Behaviour

The previous section showed how we can describe the structure of hardware modules by entities and architectures. This is not enough for making a complete hardware design. At one or another level, we need to specify logic behaviour.

As mentioned, the behaviour is specified within the module architecture. Typically, it can be done in a so–called *process*. The process is a collection of expressions which implement behaviour. One important thing to keep in mind is that all processes and instantiated modules in one architecture are running in parallel with each other. This is natural because both the instantiated modules and processes represent digital circuits which are mutually connected. A process is made like this:

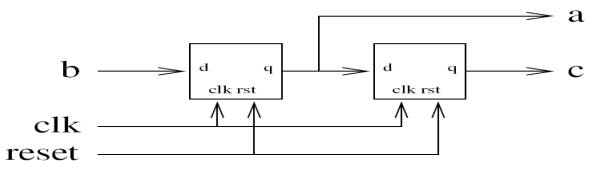

Figure 2.4: Flip–flops, FFs

```
process (clk, reset)
begin

  -- the process code comes here

end process;
```

A process begins with the specification of the so–called *sensitivity list* which states to which signals the process will react. All changes of these signals lead to the process running anew. In the example above, it is the signals `clk` and `reset` which are in the sensitivity list. This is used by the simulator so that it can recognise when to run the process anew. It is less clear what this is used for by the synthesis tools. The synthesis tools you are going to use will ignore the sensitivity list. Therefore, it is important to make a correct sensitivity list with simulation in mind, otherwise the simulation will give wrong results i.e. the synthesised design will not be as desired.

To summarise, the behaviour of a module is described by the combination of instantiated submodules and processes. If the functionality of a single module becomes too complex, typically it is split into more submodules.

Combinatorial Design

Internally within the process, we typically want to be able to specify a given digital circuit. This is done by the combination of sequential statements, boolean expressions and signal assignments.

Signal Assignments In a process, a signal is assigned a value like this:

```
eksempelsignal <= '0';
```

Sequential Statements There is a whole row of sequential statements in VHDL. It can be somewhat confusing because the result of these statements is no sequential program but the circuit structure.

Here is an example of **if**–statement which will result in the MUX shown in Figure 2.5:

```
if a = '0' then
  b <= c;
else
  b <= d;
end if;
```

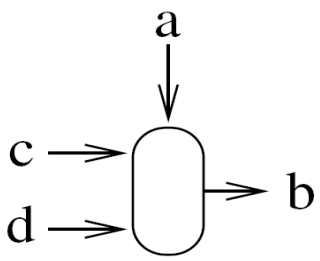

Figure 2.5: Multiplexor, MUX

Here is an example of **case**–statement which reminds on switch–case in C and Java:

```
case a is
  when '0' =>
    b <= c;
  when '1' =>
    b <= d;
end case;
```

This **case**–statement will result in the same MUX as the **if**–statement above. The advantage of using a **case**–statement becomes more obvious when a conditioning signal (`a` in the above example) is more than one bit wide. For example, if we had a 3–bit signal, there would be eight possible outcomes. In such a case, one **case**–statement with eight **when**–expressions would be much nicer than a row of nested **if**–statements.

Boolean Expressions We have certainly the possibility to implement boolean algebra in VHDL. After all, that is the basis of the language for specifying digital circuits. Here are some examples:

```
a <= b and c;
d <= d nor (e and f);
```

Flip–Flops

We often need a synchronous design i.e. the design which includes flip–flops. So, how can we make a flip–flop in VHDL?

Flip–flops (and latches) are automatically generated if VHDL is written in a specific way. Take a look at this example:

```
process (clk, reset)
begin
  if reset = '1' then
    a <= '0';
  elsif rising_edge (clk) then
    a <= b;
  end if;
end process;
```

This is a so–called synchronous process with asynchronous reset. This means that the circuit will react immediately to the reset signal if it is set high but everything else is happening synchronously with the clock signal. If the reset

signal has value '1' (high), the signal `a` will be set low. If the reset signal is not set and there is a rising edge on the clock signal (`rising_edge(clk)`), then `a` will be set to the same value as signal `b`. This happens only when the clock goes from low to high, therefore corresponding to the definition of a flip–flop. Synthesis tools will therefore make a flip–flop for signal `a`.

Someone might notice that signal `b` is not in the sensitivity list. We have already mentioned that the sensitivity list must contain all the signals to which the process needs to react. Here, it is not necessary to include signal `b` because a synchronous process needs to 'wake up' only when either reset or clock signal is changed.

It is worth mentioning that we could get arbitrarily complicated logic within an **elsif**–block and all the signals which are set here would become FFs.

One important property of the process is that the signals which are used within the process have a value from the previous time when the process was run. Let us take a look at this new example of a flip–flop:

```
process (clk, reset)
begin
  if reset = '1';
    a <= '0';
    c <= '0';
  elsif rising_edge (clk) then
    a <= b;
    c <= a;
  end if;
end process;
```

Here, we have made two flip–flops of signals `a` and `c`. The flip–flop `a` will be exactly as in the previous example. The flip–flop `c` will also be a common flip–flop but which value will `c` get? A natural thing to think is that `a` and `c` will always have the same values but this is not the case. `c` is assigned a *previous* value of the signal `a` i.e. the value `a` got in the previous cycle. This circuit is schematically shown in Figure 2.4.

Latches

It is also possible to make latches in VHDL. It can be done in the following way:

```
process (b, c)
begin
  if b = '1' then
    a <= c;
  end if;
end process;
```

Here, we have made a latch out of signal `a`. This is because we have not specified what will happen if the signal `b` is low, we have just said what happens when `b` is high. Therefore, the synthesis tools have to make a latch so that the signal `a` is held constant in case `b` is low.

Latches are rarely needed so most often something has gone wrong if the synthesis tools must introduce latches. Typically, we unintentionally forget to specify all possibilities either in an `if` or a `case` statement as it was demonstrated in the example above. To avoid a latch in this example, we can include an `else` block which sets `a` to something when `b` is low.

State Machines

State machines are a common way to make a control logic in VHDL because there is often a need for implementing some form of sequential logic. A usual way of making a state machine is shown here:

```
architecture fsm_arch of fsm is

  -- set up the new data type (nor. 'tilstand' <=> eng. 'state')
  type tilstandstype is (tilstand_1, tilstand_2, tilstand_3);

  -- state register
  signal tilstand : tilstandstype;

begin

  process (clk, reset)
  begin

    if reset = '1' then
      tilstand <= tilstand_1;

    elsif rising_edge (clk) then

      case tilstand is

        when tilstand_1 =>
          tilstand <= tilstand_2;

        when tilstand_2 =>
          tilstand <= tilstand_3;

        when tilstand_3 =>
          tilstand <= tilstand_1;

        when others =>
          tilstand <= tilstand_1;

      end case;

    end if;

  end process;

end fsm_arch;
```

This state machine is shown in Figure 2.6. First we introduce a new type for our state register. This type contains all different states we need. Then we set up the state register itself. It is a register because we assign it in an `if rising_edge(clk)` block further below.

Our `case`–statement makes a choice over the state register and performs different things dependent on the state we are in. The only thing which is done in our example is to update the state register but a real state machine will in addition do other things here. Pay attention to the `when others`-statement. It is there to cover all possible states so that we can get a defined behaviour also when we end up in an unexpected state for one or another reason.

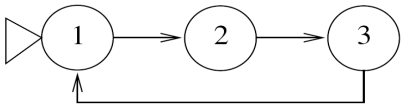

Figure 2.6: A state machine

2.2.4 Simulation of VHDL Code

It is important to simulate the circuits designed in VHDL. Even if the design can be tested out in an FPGA, there are limited debugging possibilities there so typically the errors are found through simulations beforehand.

Test benches in VHDL

Simulation is typically conducted with the use of so–called **test benches**. These are VHDL modules whose only task is to instantiate circuit designs (which are called Unit Under Test (UUT) within the test context) and test if they work as expected. This is done by setting the values for all inputs of a UUT which is followed by checking if the circuit reacts correctly to these test vectors. The entity of a test bench will not contain any signals because a test bench can not be instantiated at any other place neither can it be synthesised. A test bench is used only in a simulator so that all the possibilities and tricks of VHDL can be used here. It is not necessary to worry about whether the code is synthesisable or not. The code style in a test bench is therefore typically a bit different than that in the circuit design.

Here is a test bench for the test module we made in section 2.2.2.

```
library ieee;
use ieee.std_logic_1164.all;

-- empty entity for testbenches
entity testentitet_tb is

end testentitet_tb;

-- testbench architecture
architecture testbench_arch of testentitet_tb is

  -- declare test entity
  component testentitet
    port (
      testinput : in std_logic;
      testbuss  : inout std_logic_vector(7 downto 0);
      testoutput: out std_logic;
      clk       : in std_logic;
      reset     : in std_logic);
  end component;
```

```
  --internal signals
  signal testinput_i : std_logic;
  signal testbuss_i  : std_logic_vector(7 downto 0);
  signal testoutput_i: std_logic;
  signal clk_i       : std_logic;
  signal reset_i     : std_logic;

  begin                  -- testbench_arch

  -- instantiate UUT (Unit Under Test)
  UUT: testentitet
    port map (
      testinput  => testinput_i,
      testbuss   => testbuss_i,
      testoutput => testoutput_i,
      clk        => clk_i,
      reset      => reset_i);

  -- make clock signal (100ns period)
  clk_proc: process
  begin
    while true loop
      clk_i <= '1';
      wait for 50 ns;
      clk_i <= '0';
      wait for 50 ns;
    end loop;
  end process;

  -- press the test vectors
  test : process
  begin

    -- first reset the circuit
    reset_i <= '1';
    wait for 100 ns;

    reset_i <= '0';
    wait for 100 ns;

    -- press the test vectors here in the same way as it was done
    -- with the reset signal in the code above but this time for
    -- other signals of the circuit

    -- ...

  end process;

end testbench_arch;
```

We can see that this is a description of a test bench module with completely empty entity, no input or output signals for the module. Within the architecture, UUT is instantiated (the module we would like to test). Then, a process follows which produces the clock signal (with period of 100ns). Towards the end, there is a process which first resets the circuit and then applies the test vectors. The stimulation by test vectors is not given in this example but it is done by signal assignments and **wait** statements.

Pay attention to the use of the language constructors such as **while**–loops and **wait**–statements. These are not permitted in the synthesisable code but can be used in test benches. Especially useful is a `wait for X ns` which can

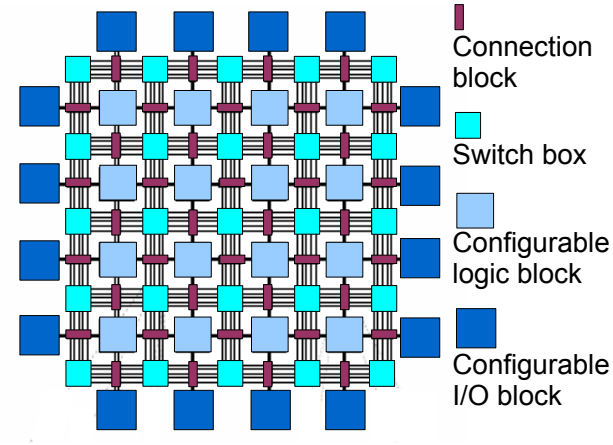

Figure 2.7: A generic architecture of an FPGA

be used to apply test vectors at different points in time. For the case of **wait**–statements, a process will be run sequentially as in a common programming language. Other useful VHDL possibilities specific for simulation from which the test benches benefit include input/output to file and screen.

A good test bench will test most possible of the situations which can occur. This can be done by manually writing a set of test vectors in the test bench which will stimulate UUT. More advanced test benches can be written so as to generate test vectors automatically with the use of a random number generator. In this way, a large number of random test vectors can be tested and, therefore, more can be covered than it would have been by hard–coding manually defined test vectors into the test bench.

Simple test benches rely on the person who performs simulation to manually examine in the simulator that the circuit reacts correctly to the stimuli from the test bench. More advanced test benches check themselves if the output of a UUT is correct or not and write the result down into the file.

2.3 Field Programmable Gate Arrays, FPGAs

FPGAs are semicustom, array–based, pre–wired digital integrated circuits ICs. Introduced in the mid 1980s when the gap between the rising design complexity and the design productivity was widening, FPGAs offered a solution in a form of arrays of reconfigurable blocks whose logic function and interconnectivity could be programmed by users. Among the chips which implement digital logic, FPGAs are somewhere between Application Specific Integrated Circuit (ASIC) and General Processor (GP). For the former, which are tailored to a specific application, computation is done in hardware, while the latter make use of silicon (Si) reusability by sequentially performing a sequence of instructions – a program – on the same hardware. FPGAs can also be programmed through the process of configuration of its logic blocks and their interconnectivity. They

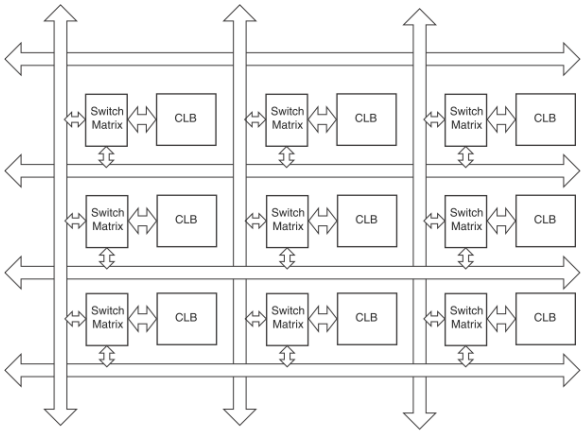

Figure 2.8: General architecture of Spartan–6, from [7]

can be programmed many times i.e. reconfigured many times, abiding to the requirements for a Si reusability as GPs do. However, in FPGAs a computation is performed in hardware so that they benefit from the same advantages as ASIC designs do, but avoiding at the same time the high production costs which accompany ASIC design. Although high production costs prevent them from being widely used, they have still found areas of application in certain fields. However, due to the possibility to be reprogrammed, they have become a valuable asset for prototyping because of the lower costs and time of the prototype production.

Figure 2.7 shows a schematic view of the general FPGA architecture with four main elements: Configurable Logic Blocks (CLBs), configurable Input/Output (I/O) blocks, switch boxes and connection blocks. The logic implemented in an FPGA chip is dependent on the configuration of the CLBs and interconnectivity realised through switch boxes and connection blocks. Basic components of an FPGA can be implemented in various ways and exact implementation is mainly dependent on the manufacturer and the concrete FPGA family. A widespread type of FPGAs are static Random–Access Memory (RAM) (SRAM)–based FPGAs whose CLBs are implemented as Look–up Tables (LUTs) in SRAM cells. A LUT can be pictured as a small memory block. They store a small amount of data which can be accessed by immediately addressing the data location. In that way, LUTs can replace processing units and save the time needed for the computation. You will work with the SRAM–based FPGA chip by Xilinx which comes from the Spartan–6 family.

2.3.1 Spartan 6

Spartan–6 [5] is the latest product from the Spartan family which is known as a low–cost family from Xilinx. Manufactured in 45nm technology, it has also been optimised for a low power consumption performing savings of up to 50% in

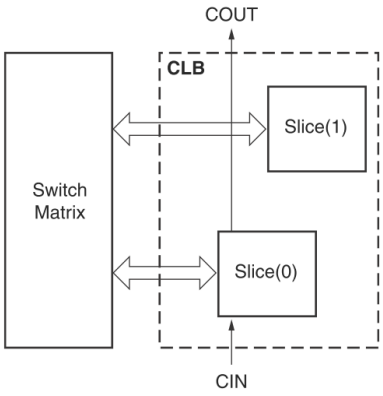

Figure 2.9: A CLB of Spartan–6 and its connection to the switch box: actual connections are between the slices within the CLB and the switch box, from [7]

comparison to its 60nm predecessor Spartan–3A. Figure 2.8 shows a schematic view of the Spartan–6 reconfigurable texture. A more detailed view is shown in Figure 2.9 for the connection between a CLB and a switch box.

In Xilinx FPGA technology, the logic is organised in CLBs. Each CLB is divided into so–called slices, see Figures 2.9 and 2.10. A slice in general represents a group of Look–up Tables and accompanying Multiplexors and flip–flops which make possible the realisation of the desired sequential or combinatorial logic. It is possible that slices contain some additional circuitry which makes them better suitable for the implementation of arithmetic operations or the use as distributed RAM and shift registers, for example.

I/O resources are manufactured in SelectIO technology and are grouped in I/O interface tiles. Beside I/O blocks, each tile contains logic blocks and buffers.

Interconnects play an important part and in Spartan–6 there are four different types as shown in Figure 2.11. Fast interconnects are used in simple functions to avoid unnecessary usage of resources otherwise used for implementation. Single interconnects are used for the connection with immediate neighbours, while double interconnects do the same for every other tile. Quad interconnects provide the connection with a fourth tile in all four directions, something like the long lines in previous generations.

There is often a need for the design implemented in FPGA to make use of certain amounts of memory. In order to reduce the time of accessing the data stored in the memory, memory can be placed on the FPGA chip. On one side, it is possible to use LUTs for that purpose. LUTs are used as data storage and combined into memories of the desired size. Such usage is known as 'distributed memory' because the memory which is implemented in LUTs is actually distributed across the chip area as are the LUTs which are used for its implementation. On the other hand, Xilinx has also provided another type of the on–chip memory – Block RAM (BRAM). As the name suggests, these are dedicated memory blocks. BRAM can be accessed through dual ports. The capacity is usually of several kB and an FPGA can contain several blocks of

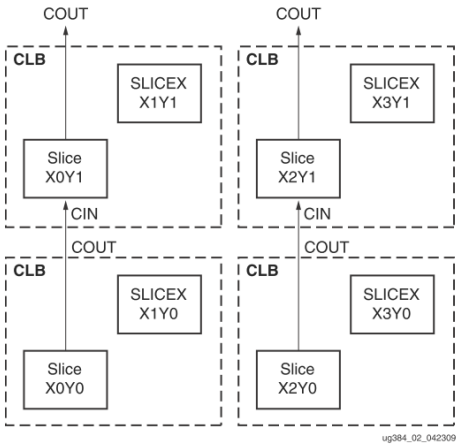

Figure 2.10: The placement of the CLBs and the pertaining slices into a matrix, from [7]

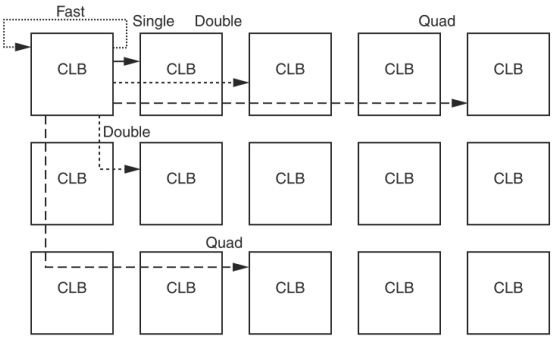

Figure 2.11: Types of interconnections in Spartan–6, from [7]

BRAM. Spartan–6 family contains up to 18kB of BRAM in blocks of 9kB. More on BRAM in Spartan–6 can be found in [6].

There are many other features of Spartan–6, like handling of clock resources, for example, which are examples of how clever design an implementation can yield desirable results with respect to speed, power consumption and similar requirements. For those interested in the details of the Spartan–6 architecture, more can be found in the documents provided by Xilinx on its official site.

2.3.2 S5LX16 Development Board

For the development of FPGA–based applications, a range of development boards exists. The one you will work with is produced by Avnet and is shown in Figure 2.1. It contains one Spartan–6 chip, XC6SLX16-2CSG324C, and other resources which enable the user to access the FPGA and test its operation. The board is self–powered by a rechargeable battery which is recharged every time the board is connected to the PC. The connection with the PC is a Universal Asynchronous Receiver/Transmitter (UART) serial communication via a Universal Serial Bus (USB) cable. On the PC side, the communication with the board is realised through a virtual COM port configured for the following settings:

- 115200 bits per second
- 8 data bits
- no parity
- 1 stop bit
- no flow control

The power switch is SW1 and it has to be in the position **on** before the board is connected to the PC. When a USB cable is connected between the board and the PC, the diode D18 is lit up. If the battery was disconnected from the board connector, the diode D16 will be blinking so make sure to connect the battery before you start using the board. When the FPGA is being configured, the diode D11 will blink blue and then remain lit up blue after the configuration is completed.

2.4 Design and Implementation in FPGAs – a Walk through the Xilinx ISE Design Suite

2.4.1 Xilinx ISE

The road from the design to the implementation in FPGA is not a simple, one–step process. It takes several steps each of which is followed by the verification of the design. Figure 2.12 shows a flow diagram of these steps according to the specification by Xilinx. Other FPGA–vendors also provide their own specification but, in essence, the steps are as described here.

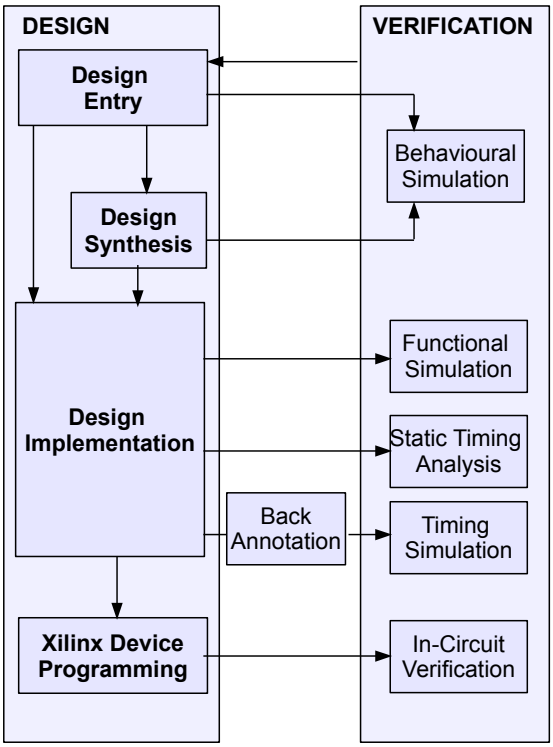

Figure 2.12: FPGA Design Flow (adapted from iseguide on www.xilinx.com)

Design Entry – consists of the source files for the design modules and the constraints the design should obey (user constraints, timing constraints, area constraints, pin assignments). Source files may be of different types but for the assignments you will make your design in VHDL (.vhd files).

Design Synthesis – generates a netlist for your design. A netlist is a description of your design in a form of a list of the design components, component attributes and the interconnectivity between them. For the generation of a design netlist, a Xilinx Synthesis Technology, XST, is used. As a result, the netlist for your design is saved in a specific format – an .ngc file.

Design Implementation – implements the netlist provided in an .ngc file in the form which corresponds to the particular FPGA chip so that the chip programming can be performed for the available FPGA resources. Design implementation is performed through three processes: Translate, Map and Place and Route. The Translate process merges the netlist and the design constraints and produces a logical design reduced to Xilinx primitives. The latter is given in a form of Xilinx native generic database file, .ngd file. The Map process produces a native circuit description file, .ncd file, which maps the logic design to physical components of FPGA such as CLBs and I/O blocks. The Place and Route process places the mapped design on an FPGA and routes the interconnections between design components. It produces an .ncd file with the design placed and routed for the actual FPGA.

Xilinx Device Programming – generates a .bit programming file out of the .ncd file produced in the Place and Route process. The programming file provides the information for the configuration of the resources on the physical chip.

Figure 2.12 also shows various types of design verification dependent on the available format of the design. Verification at a high abstraction level (behavioural simulation) is fast, but it may not uncover all the timing issues which may occur when the design is implemented on the chip. Verification at a low abstraction level (timing simulation) is slower but more accurate. Different types of the design verification have already been explained in Section 2.2.

Xilinx ISE provides you with the tools to design and implement your design on a Xilinx FPGA chip. It is an extensive and rather complex tool and for more information we refer you to the product documentation. A tutorial about its use is also added to the folder with useful files on the course It's learning page [4]. It may be helpful if you consult this document during your work on the assignments as well as Xilinx database for FAQs and forums devoted to FPGA design.

In order to introduce you to the ISE Project Navigator and show how to perform the described steps, we present you with a simple task – to design and implement an incrementer within ISE Design Studio. As a support, screenshots are provided for each step. We advise you that you perform the described steps yourself within ISE environment.

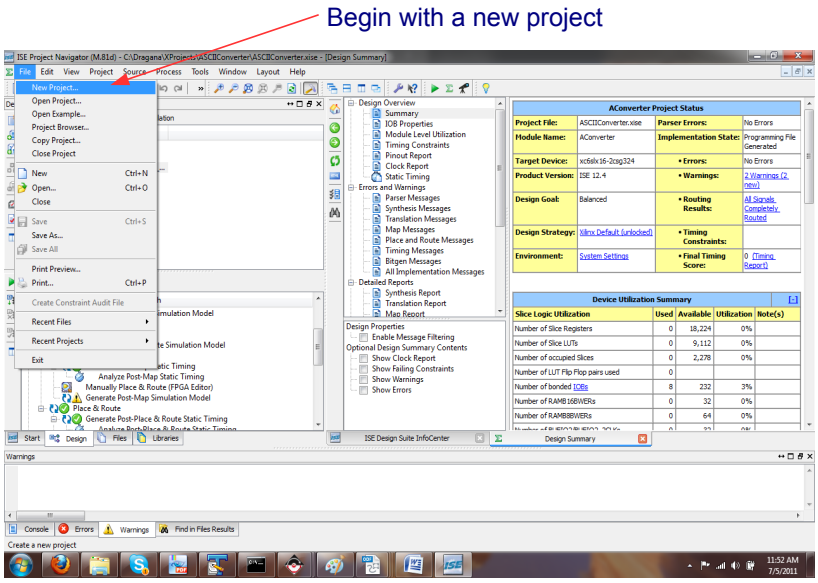

Figure 2.13: Opening a new project in ISE Project Navigator

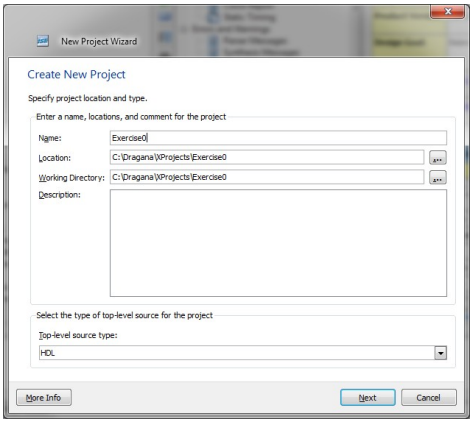

Figure 2.14: A New Project window with the specification of the project name and the location for the project files

VHDL Design

Open the ISE Project Navigator from the Start menu of your computer. In ISE, a design is implemented as a project so choose **File** → **New Project** option from the menu bar on top as shown in Figure 2.13. A window will open in which you will be asked to provide the name for your project and the location for the corresponding files, see Figure 2.14. For the top–level source type leave the offered option of HDL as you will make your design in VHDL. Click **Next**.

The window opens in which you are asked to specify a Xilinx chip you will be using and the settings for your project. As Avnet S6LX16 development board contains a chip from the Spartan 6 family i.e. XC6SLX16-2CSG324C, your chosen options should be as in Figure 2.15. Mark that the speed grade is changed to -2 from the originally offered -3. As a synthesis tool you will be using Xilinx Synthesis Technology, XST, so leave the offered option chosen. For the simulator within your project choose **Modelsim–SE VHDL** and VHDL as a preferred language. Click **Next**. A summary of the project settings appears as shown in Figure 2.16. Click **Finish**.

In the window which opens, see Figure 2.17, there is a **Design** pane in the top left corner. In the **Implementation** view, as is originally chosen, this pane shows all design files within the project. Design files are ordered hierarchically according to the entities within the design they contain. For each project, there is one top–level entity which contains all the remaining ones. At the moment, only the chip is symbolically shown and the folder with user library modules is empty.

To add a source file, right–click on the chip symbol and click on the **New Source**, see Figure 2.18. A window like the one in Figure 2.19 opens. Here you can choose the type of the source file to be included in the user library for your project. Mark **VHDL Module** because VHDL was chosen for the design in the project settings in the beginnings. Choose a name for your module. In this simple exercise, there will be only one module which will be, therefore, a top–level module. So, we suggest you name it simply a – toplevel. Leave the

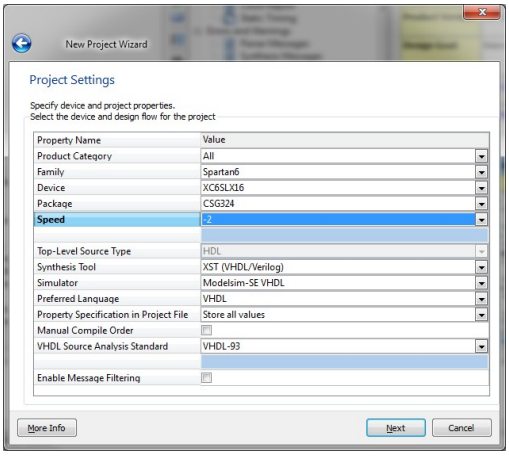

Figure 2.15: Specification of the FPGA chip to be used in the project and the project settings

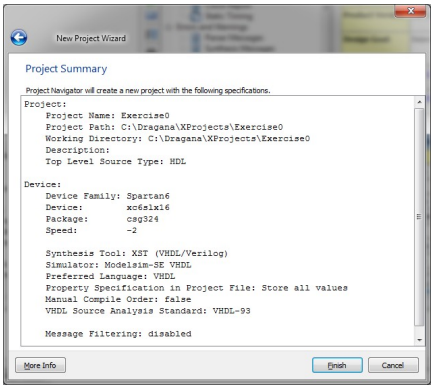

Figure 2.16: A summary of the project settings

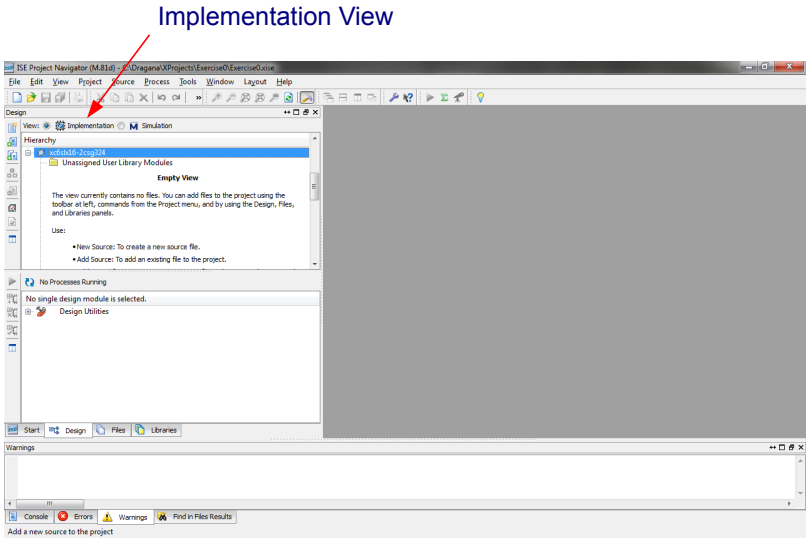

Figure 2.17: New Project opened with no design sources

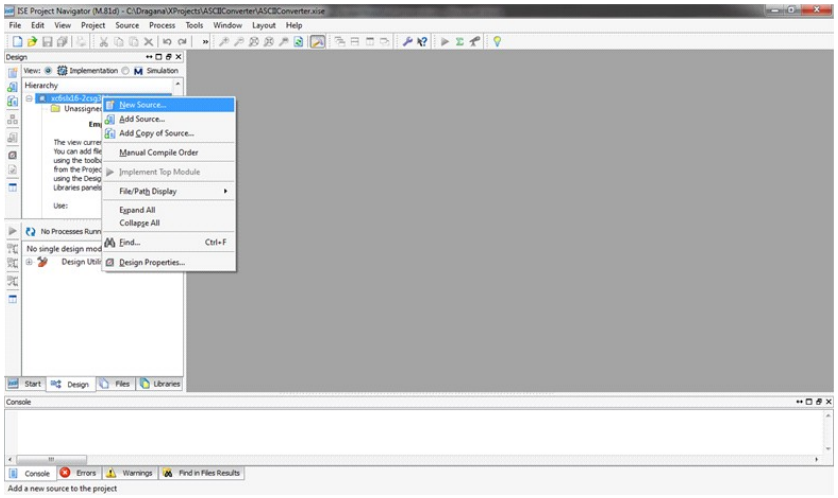

Figure 2.18: Adding a new source

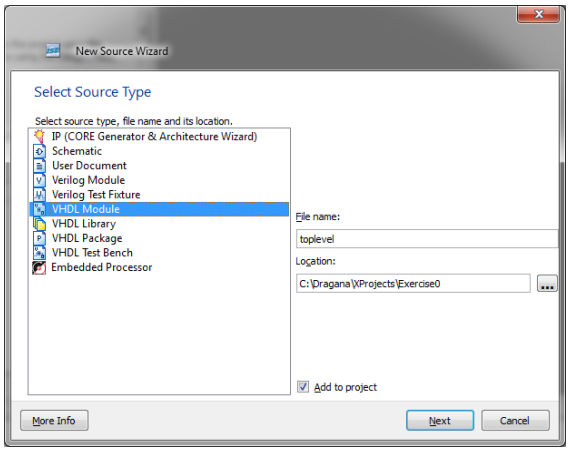

Figure 2.19: Choosing type, name and the location to be saved at for the new source file

suggested location for the toplevel source file to be saved in the project folder. Click **Next**.

In the window which opens, define the interface of your toplevel entity – the directions and types of the port signals. A simple design considered in this example has four interface signals as shown in Figure 2.20. Input signal **clk** stands for the input clock signal, input signal **reset** is a reset signal. When its value is high, all the output lines should be reset, in this case to '0'. The input for the incrementer is provided on a 32–bit input bus, **bus_ in**. The result of the incrementer operation is produced as an output on a 32–bit bus **bus_ out**. If you do not specify interface signals for your module, you can do that in the corresponding .vhd file. After making the choice, click **Next**.

A summary of your specification for the new module opens in a window, as shown in Figure 2.21, where you can check once again if everything is as you want. Click **Finish** if you agree.

ISE Project Navigator environment now shows a newly added module in the **Design** pain in the top left. Below it, in the **Processes** pane, a list of available processes for the design is shown when you mark the entity as shown in Figure 2.22. To the right, a VHDL code for the newly added module is generated based on the specification you have provided. It is a skeleton which leaves you space to implement the architecture of your module. Add the code as in Figure 2.23 which implements the behaviour of the incrementer module:

```
entity toplevel is
    Port ( clk      : in  STD_LOGIC;
           reset    : in  STD_LOGIC;
           bus_in   : in  STD_LOGIC_VECTOR (31 downto 0);
           bus_out  : out  STD_LOGIC_VECTOR (31 downto 0));
end toplevel;

architecture Behavioral of toplevel is

begin
```

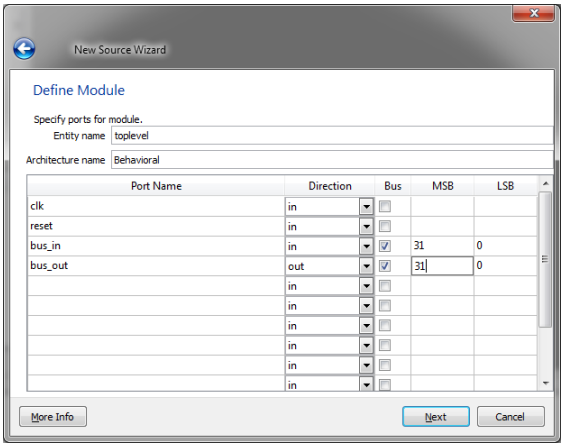

Figure 2.20: Specifying the module interface

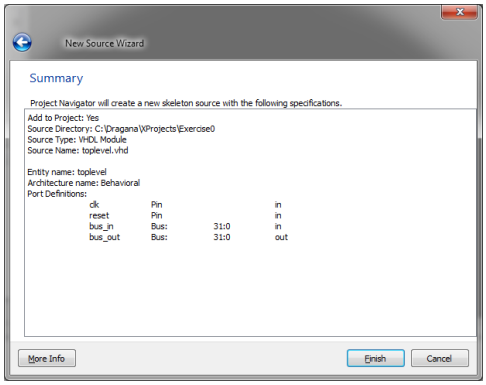

Figure 2.21: A summary of the module specification

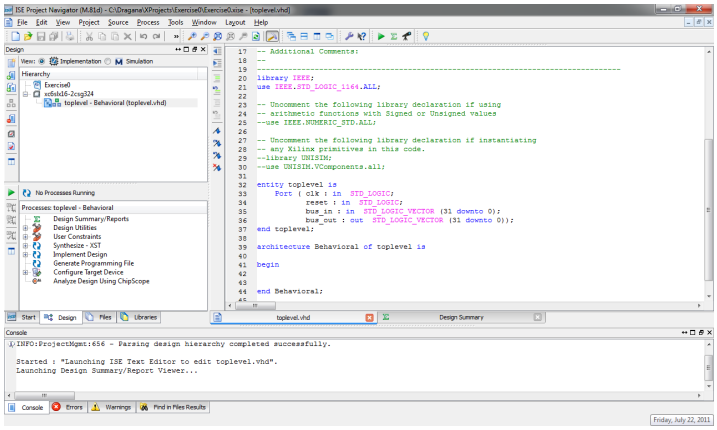

Figure 2.22: A new module added within the design hierarchy

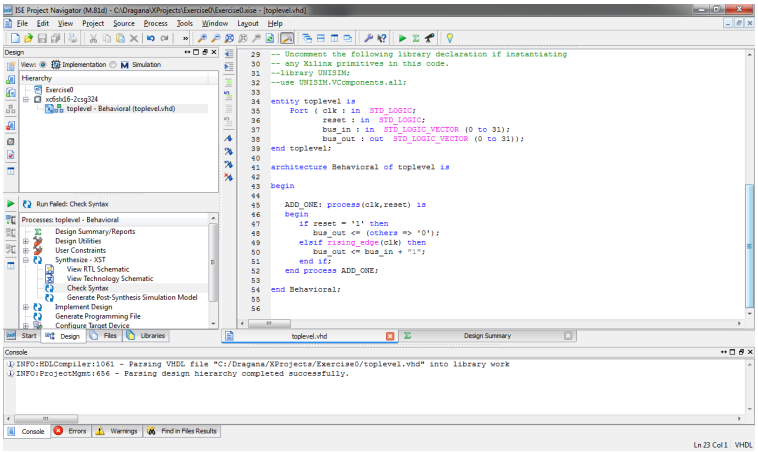

Figure 2.23: VHDL code which implements the architecture of the incrementer module

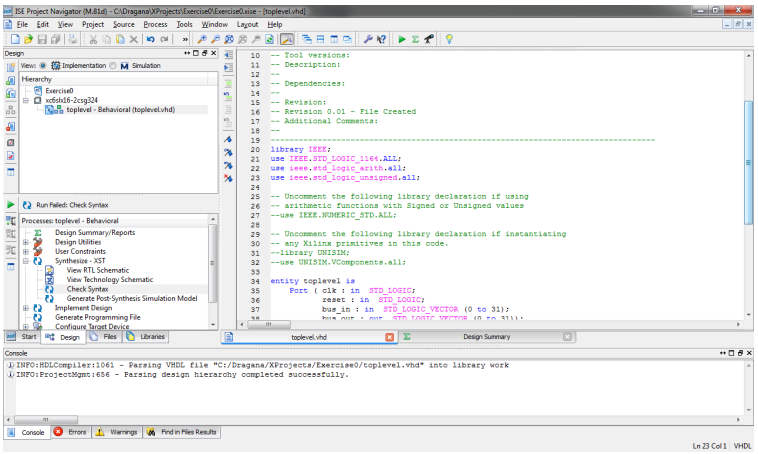

Figure 2.24: The libraries which need to be included in order to make the code from Figure 2.23 error–free

```
    ADD_ONE: process(clk,reset) is
    begin
        if reset = '1' then
                bus_out <= (others => '0');
        elsif rising_edge(clk) then
                bus_out <= conv_std_logic_vector(unsigned(bus_in) + 1,32);
        end if;
    end process ADD_ONE;

end Behavioral;
```

It is as follows: when **reset** signal is high, it resets all lines of the output bus to logic '0'; otherwise, on the rising edge of the clock signal, it increments the value on the input bus by one. For the sake of simplicity, no additional checks have been implemented for the maximum value to be represented on the input lines.

Beside the code given above, you also need to include some libraries in addition to those which are automatically included when a .vhd file for the new module is created by the ISE Project Navigator. Figure 2.24 shows which libraries need to be included.

Before you proceed to the design synthesis step, you need to be sure that your VHDL code is free of syntax errors. To invoke the syntax check, mark the module in the Design pane (in this simple case it is only the toplevel module) and in the Processes pane click on **Check Syntax** on a subtree below a **Synthesis - XST** entry, see Figure 2.25. If the code is error–free, the message like the one in the **Console** pane at the bottom is shown. In case there are any syntax errors in the code, you may view them in the **Errors** pane at the bottom.

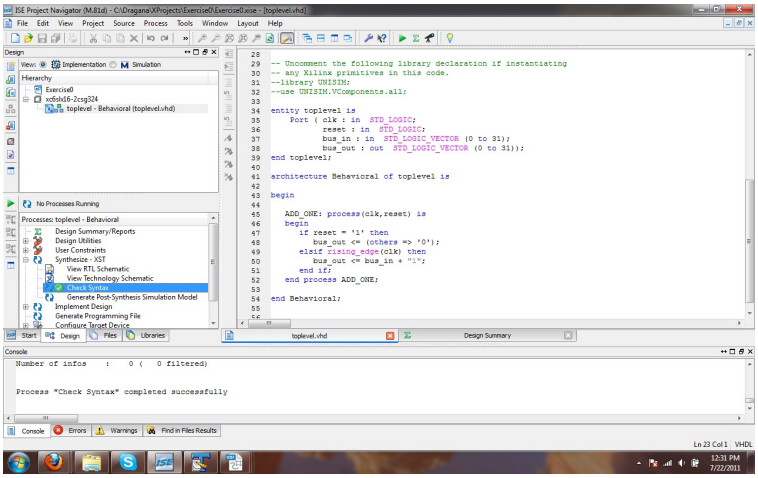

Figure 2.25: Checking the syntax of the VHDL code pertaining to the module toplevel

Behavioural Simulation – ModelSim

Error–free syntax of a VHDL–code does not mean that the code will lead to the generation of hardware which behaves in the way you would like it to. To check if it is so, you need to simulate the behaviour of the module architecture described by your code. The module behaviour can be simulated so that any 'misbehaviour' can be detected at this early design phase and accordingly corrected by re–writing the piece of VHDL code in question. For simulation, you will use ModelSim as specified at the beginning of your project. In Figure 2.12 it is shown that for a design entry, in our case described in VHDL, a behavioural simulation can be performed.

Behavioural simulation means the simulation of the VHDL code in its original form with the assumption that all the components are perfect and with no delay. It is a fast simulation and it can reveal many types of functional errors in the circuit. Behavioural simulation can be run also only by ModelSim and then no access to the synthesis tools is needed. A common approach to simulate a given circuit is to make a test bench as described in Section 2.2.4. Once more, a test bench is a VHDL entity which has an empty port description and whose architecture instantiates the circuit design which will be tested. This entity is known as a **Unit Under Test, UUT**. A test bench sets the test vectors and the circuit response can be checked either manually in a 'waveform viewer' or automatically by the code in the testbench itself. A test bench is not synthesisable and therefore it can not get use of the whole of the VHDL language.

Test benches can be written from scratch, but Xilinx ISE provides a support for their generation as well, at least to the point when test vectors need to be specified. Right–click on your toplevel in the Design pane and choose **New**

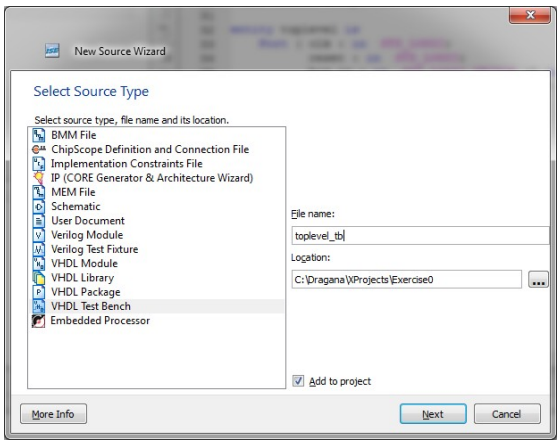

Figure 2.26: Creating a new test bench file

Source. In the window which opens choose VHDL Testbench for the source type, see Figure 2.26. Choose the name for new source file as toplevel_tb. Click **Next**. In the next window you can choose with which entity from your design the testbench will be associated, see Figure 2.27. In our case there is only one – toplevel entity so select it and click **Next**. The summary of your testbench appears as in Figure 2.28. Click **Finish**.

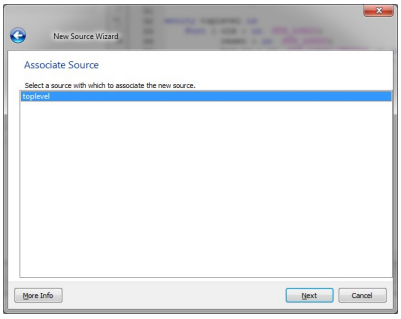

Figure 2.27: Choosing the entity which will be tested within the test bench

The generated code for the test bench is shown in the left pane of the ISE Project Navigator. Take a look at the code, the entity has no interface input / output signals, it cannot be synthesised. Further, mark that the UUT i.e. our toplevel module is instantiated as a component, its input and output signals mapped to signals within a test bench, see Figure 2.29. Also, mark the process for clock generation. The result of this process is a signal which will play the role of the clock signal for the testing purposes. The toplevel module is tested within the process below the clock–generating process in Figure 2.29. Its inputs are assigned certain test vectors at certain times. In Figure 2.30 you can see how this is done – we have chosen a few test vectors for this purpose.

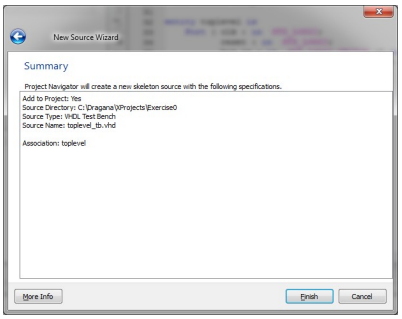

Figure 2.28: A summary information on the creation of a test bench file

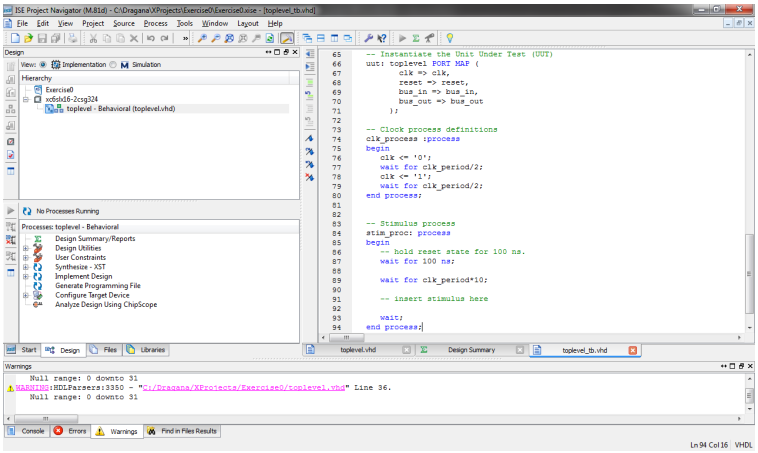

Figure 2.29: The generated skeleton for the testbench

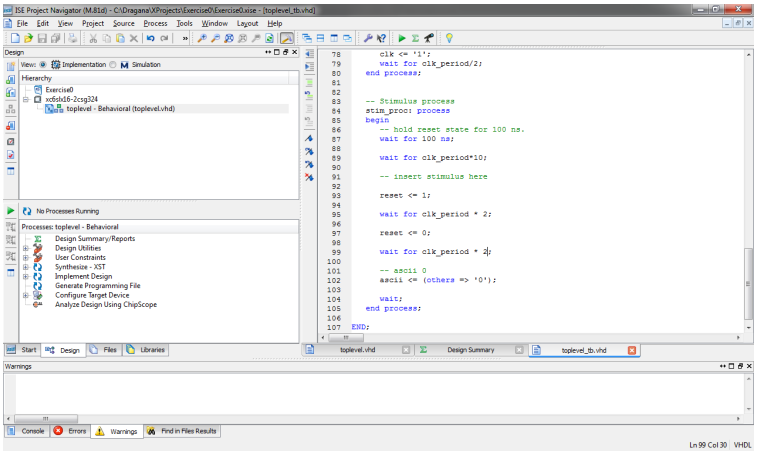

Figure 2.30: An example of the assignment of test vectors to the inputs of the UUT

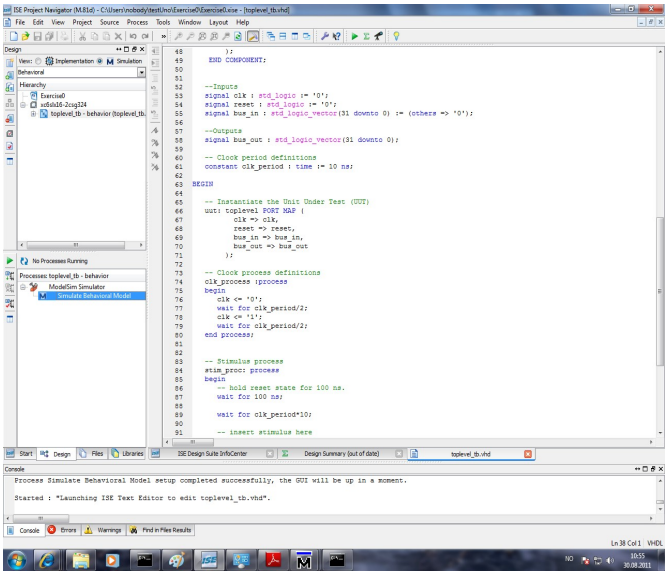

Figure 2.31: Invoking ModelSim from ISE environment

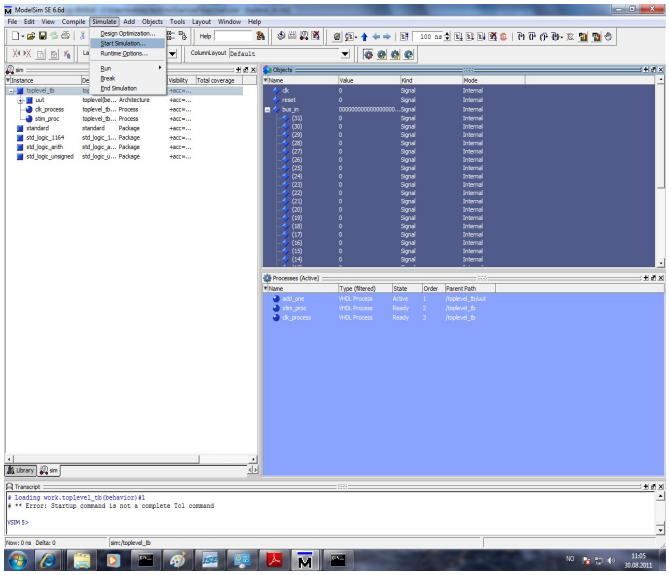

Figure 2.32: Starting simulation in ModelSim

The ISE software has a full integration with the ModelSim simulator. Therefore, you can open the ModelSim simulator by clicking on the **Simulate Behavioral Model** in the **Processes** pane when the toplevel test bench is marked, see Figure 2.31.

In the **Library** view, you get the overview over all libraries and logical structure. Your circuit will be in the 'work' library after compilation. VHDL files have to be compiled in the special sequence because of the dependencies between the files. The compilation sequence is specified through the menu choice **compile** → **compile order**. Here you can set the sequence by yourself or try **Auto Generate**. This will compile all files and find the dependencies but it will do so only if all the files are error–free. For your simple design of the incrementer no specification for the compilation order is needed.

Compile the source files with the menu choice **Compile** → **Compile**. When you have more files to compile for simulation, you will use **Compile** → **Compile All** option. If all the files have been compiled without error, you may begin with the simulation. Choose your test bench in the list over libraries (card 'Design'). Remember that all your design modules are placed in the library 'work'. Menu choice **Simulate** → **Start Simulation** starts the simulation see Figure 2.32.

In the simulation mode, you will get the list of all the component instances in the workspace overview to the left. By clicking on one particular instance (for example your testbench), a list of all signals in the current instance is acquired (in the object window). What is desirable during simulation is to get the graphic overview, a waveform, over the changes of the signals in the design during the simulation run. This is set up in the following way:

- In the workspace overview, choose the instance with the signals you would

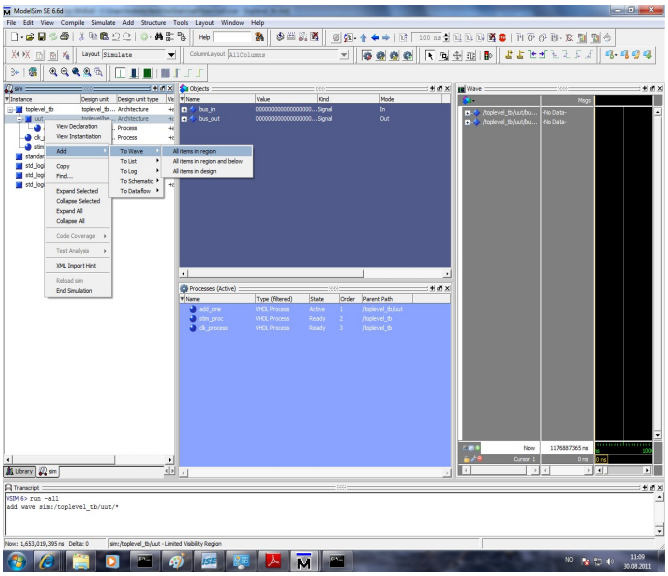

Figure 2.33: Adding signal waves in ModelSim

like to examine.

- Right–click in the object window and choose **Add to Wave** → **All Items in Region** see Figure 2.33. All signals in the chosen instance are then added in the wave window which comes up to the right.
- Run simulation by writing run in the console window or by pressing the corresponding button in the tools line.

Add all the signals in the test bench to the wave window. You will get something similar as shown in Figure 2.33. Run simulations until you are certain about that the incrementer works as it should. Figure 2.34 shows one part of the simulation results, while the position of the cursor in Figure 2.35 shows how the output bus changes at the rising edge of the clock signal. Although in the simple design for the incrementer no subcomponents are present beside the toplevel, keep in mind that it is also possible to examine the signals in the subcomponents.

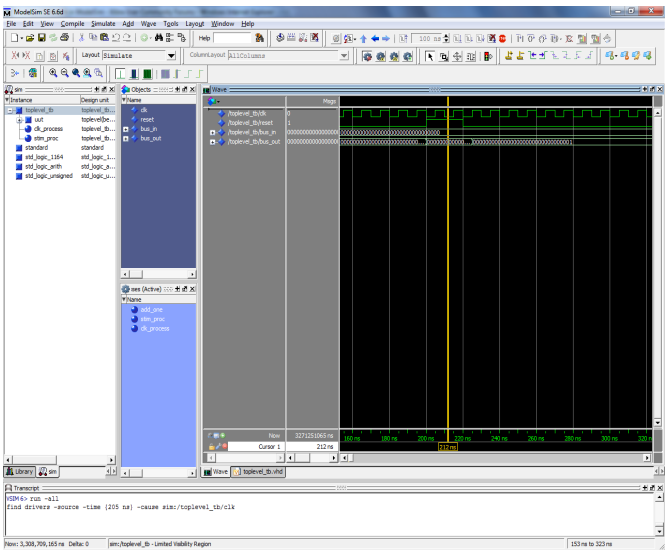

Figure 2.34: One part of the resulting simulation waves

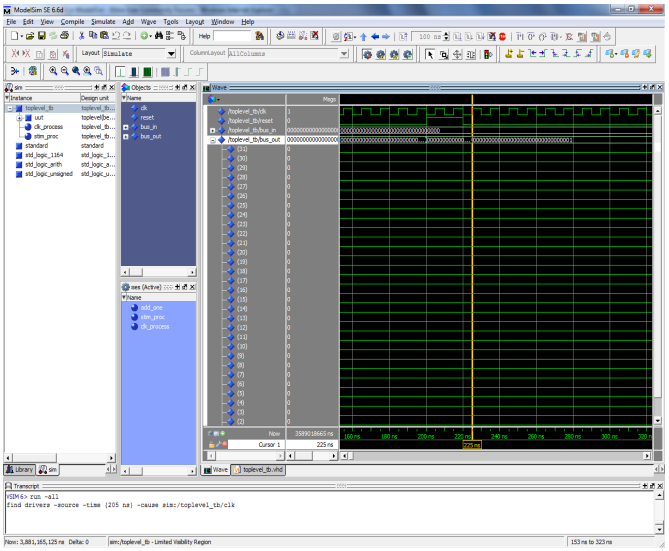

Figure 2.35: The change of the output bus lines at the rising edge of the clock signal

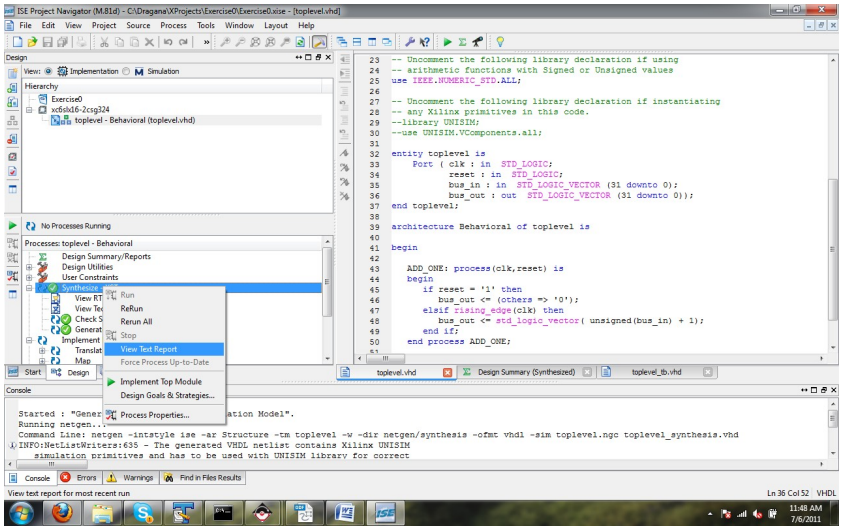

Figure 2.36: Invoking an XST tool within ISE Project Navigator

Synthesis

Now that you know that the syntax of your VHDL code is error–free and that the behaviour of the hardware to be generated out of your VHDL code is as desired, you can proceed to the synthesis step which creates a netlist out of your design in VHDL. You will use Xilinx Synthesis Technology – XST synthesis tool. If you have not invoked a syntax check beforehand, XST will do that automatically thereby preventing synthesis of any code which is not error–free. Figure 2.36 shows ISE Project Navigator after synthesis of your design described in VHDL has been performed. The report can be viewed in the **Console** pane at the bottom or, as shown in Figure 2.36, by clicking on **View Text Report** on a pop–up menu for XST, when the synthesis report opens in the pane to the right. You are advised to go through it in order to understand how your design has been transformed from the VHDL–described level to the so–called Register Transfer Level, RTL.

The generated netlist containing both – logical design and constraints (which were none in the incrementer example) is saved in the .ngc file in the project folder. The content of the file can be interpreted into an understandable form for you by clicking, for example, on **View RTL Schematic**, so that your design at the RTL can be viewed in the right pane of ISE Project Navigator environment, see Figure 2.37.

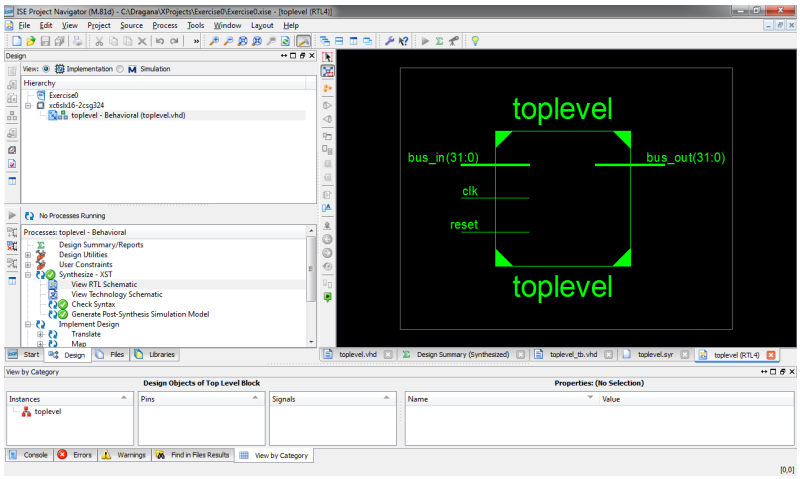

Figure 2.37: Viewing the design at the register transfer level, RTL

Implementation

As mentioned, the implementation of your design aims at producing a bitstream in a form of a .bit file which can be used for the configuration of the FPGA chip. Now we shall go through the three steps of this process which were invoked during implementation.

First, the generated logical design is translated into an equivalent description only expressed with Xilinx primitives. The design expressed with Xilinx primitives is kept in an .ngd file within the project folder. The **Map** process mapped the logical design from the .ngd file into available resources on the FPGA chip for which the project was setup. By clicking on **View Text Report** on the pop-up menu pertaining to the **Map** process in the **Processes** pane, the report is opened in the right pane. Figure 2.39 shows one segment of this report where the summary on the slice logic utilisation and distribution is shown. You are advised to go through the report and learn about the usage of individual components of the FPGA for your design – the CLBs, LUTs within them, I/O components. The data connected to your design at this level are kept in an .ncd file in the project folder. It physically represents the design mapped to the components in the Xilinx FPGA.

The step **Place & Route**, as the name suggests, places and routes the design in the way it will be implemented on an actual FPGA chip. In other words, the FPGA chip is configured based on the design generated in this step. The process itself uses the data from the .ncd file generated in the **Map** step and generates another .ncd file which corresponds to the placed and routed design and which is directly used for the generation of the configuration bitstream for the FPGA chip.

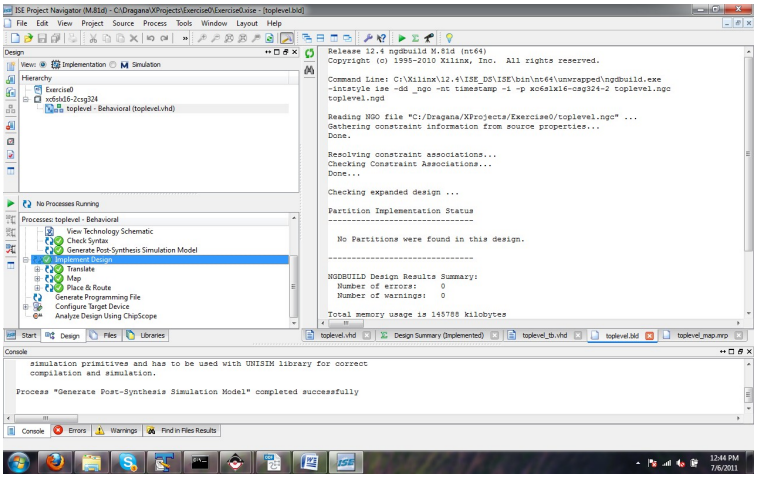

Figure 2.38: **Implement Design** step for the toplevel module and the generated report

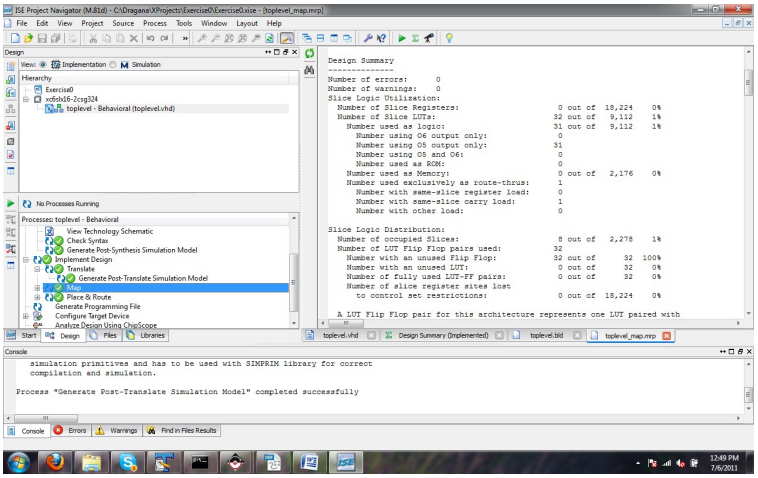

Figure 2.39: **Map** step within the design implementation and the generated report

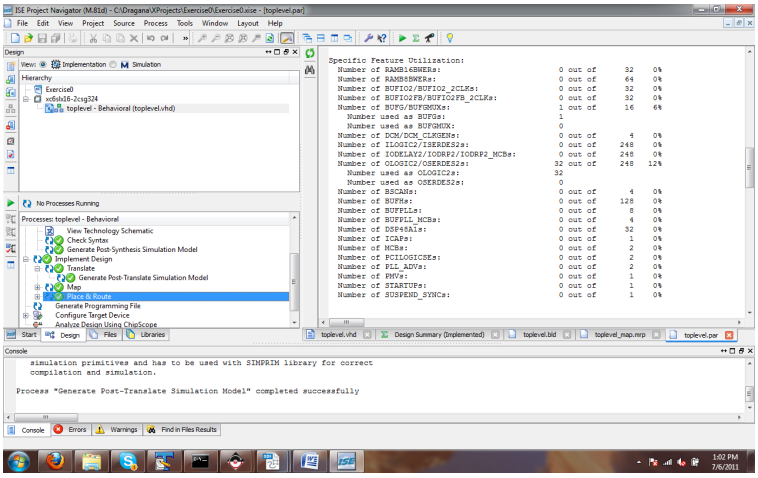

Figure 2.40: **Place and Route** step for the toplevel module and one part of the generated report

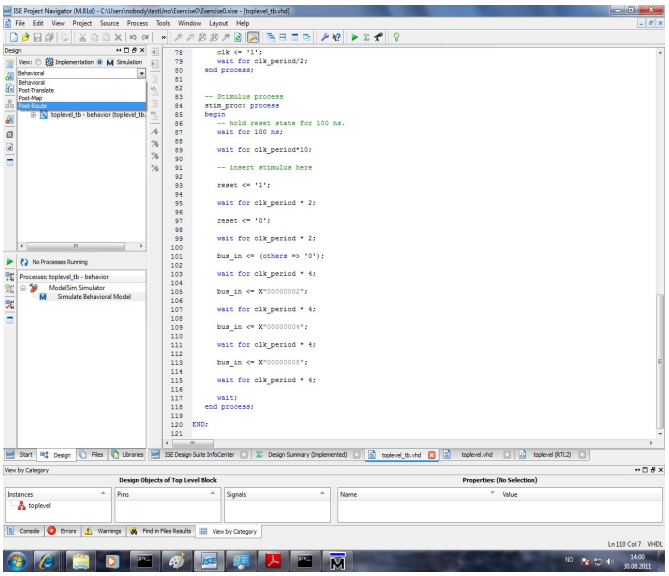

Figure 2.41: Choosing the timing simulation within the ISE environment

Timing Simulation – ModelSim

A circuit which is synthesised for a specific FPGA will always have certain delay. Different paths through the circuit have different delays. This can affect the circuit behaviour and lead to errors which are not possible to discover by behavioural simulations. After a circuit has been synthesised by the Xilinx ISE, it is, therefore, important to simulate the circuit anew with the help of the so–called timing simulation. It is also performed by the ModelSim, but this time based on the circuit description with the timing information generated by Xilinx ISE and not directly on the original VHDL files. Timing simulation is much more time–demanding than behavioural simulation and, therefore, it should come in addition to the behavioural simulation and not as a replacement for it. You can invoke the timing simulation from the ISE environment in the same way as behavioural simulation, the only difference being that you need to choose **Post–Route** item in the drop–down list which corresponds to the **Simulation** view in the **Design** pane as shown in Figure 2.41.

Therefore, in Xilinx ISE you can start timing simulation in ModelSim in the following way:

- Make sure that the test bench you have previously made is added to the project.
- Choose 'Post–Route Simulation' in the source window.
- In the process window you have to right–click on 'Simulate Post Place & Route Model' (under 'ModelSim Simulator') and choose 'Run'.
- Then you are coming directly in the simulation mode in ModelSim.

Run the simulation and check if the functionality is still correct. Pay attention to the fact that not all signals are changed at the same time with the clock signal but first after a little delay.

Generating the programming file

The configuration bitstream which is used for programming the FPGA chip is generated in a form of a .bit file within a process invoked by clicking on a **Generate Programming File** item in the **Processes** pane. The processes for the generation of the configuration files for chip–programming devices can be further invoked (see the list in the **Processes** pane) dependent on the concrete device which is used for programming the chip. However, as mentioned in the beginning of this chapter, for this purpose you will use Avnet Programming Utility which uses the generated .bit file and transfers it to the board containing Spartan 6 chip over the USB cable. Therefore, the walk through ISE Project Manager is completed with the generation of the programming file.

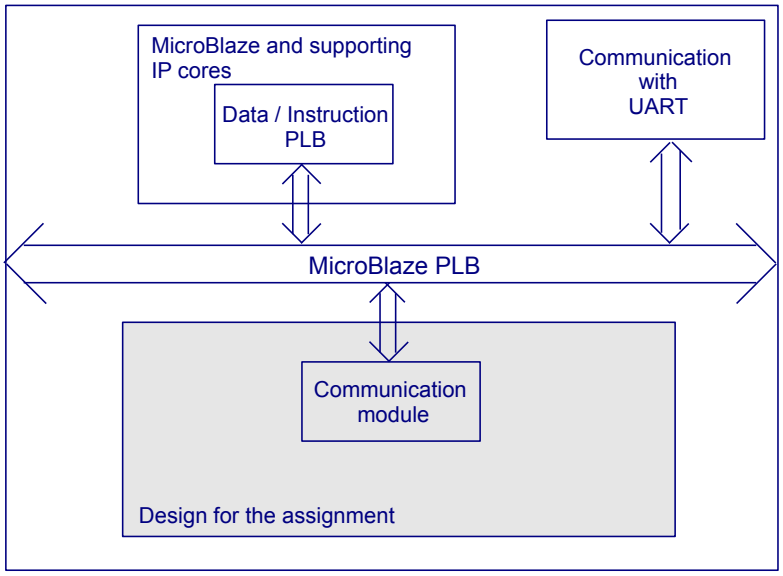

Figure 2.42: The relation between the design created for the assignment and the embedded platform within which it is to be implemented.

2.4.2 Xilinx EDK – Designing an Embedded Platform

The design which you will make for each of the assignments will not be implemented on FPGA chip as a sole design. It will be implemented as a part of the embedded platform which is run by a Micro Blaze soft processor. Figure 2.42 shows a schematic view of the design which will be implemented on an FPGA. The design you are going to make for each assignment is depicted as a shaded box which contains a communication module beside other modules dependent on the concrete assignment. The purpose of the communication module is to provide a correct communication between your design and the MicroBlaze PLB bus. The PLB bus is controlled by a MicroBlaze soft processor and one of the IP cores also connected to it is a communication module for UART through which the communication with the PC is established thereby enabling you to interact with the program running on the processor core you will implement on an FPGA.

For the assignments, you will be given a set of support files which will contain design for some of the modules for the assignment. The communication module will be among them. It will be left for you to connect the given modules and the modules you will design yourself in a correct way so that your design performs as desired. Further in this section we present you with the design of an embedded platform and show you how to include in it the incrementer from the previous section.

Start ISE Project Navigator from the Windows Start menu by clicking on **Start → Xilinx ISE Design Suite 12.4 → ISE Design Tools → Project**

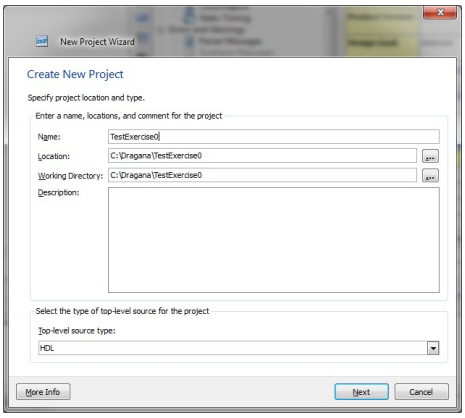

Figure 2.43: Opening new project in ISE Project Navigator

Navigator. Click on the menu item **File** → **New Project** and in the window which opens, find the location where you would like to save your project files and choose the name for the project as, for the example in Figure 2.43, TestExercise0. Click **Next**.

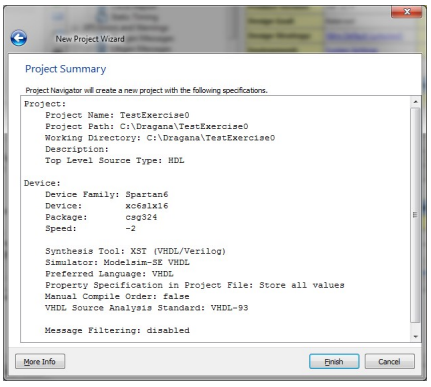

Figure 2.44: A summary of the new project in ISE Project Navigator

Choose the project settings as in Figure 2.15. Click **Next**. A summary of the project appears, see Figure 2.44. If you agree with the provided information, click **Finish**.

The new project opens in the default Implementation view, as in Figure 2.45. Right–click on the design as shown in Figure 2.18 before and in the window which opens choose **Embedded Processor** for the Source Type in the left pane. Name the system, for example **system**, as shown in Figure 2.46 and accept the offered path for saving your new source. Click **Next**.

A window opens in which the summary of the system is shown as in Figure 2.47. Pay attention to the notification at the bottom of the window which says

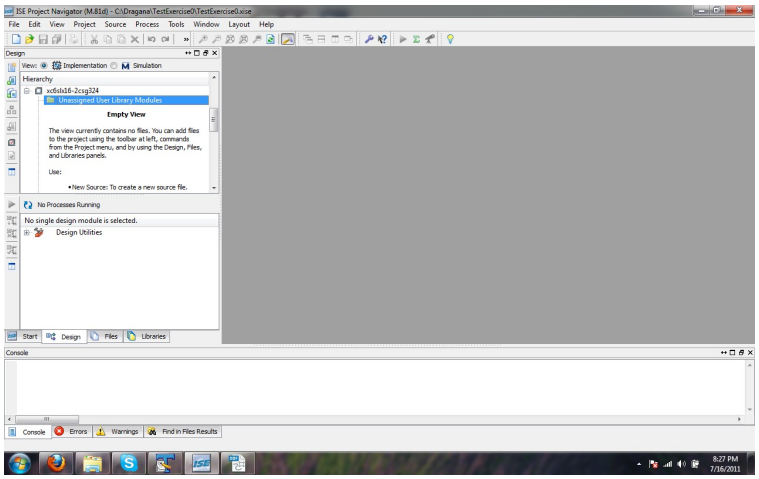

Figure 2.45: A new project in ISE Project Navigator with no assigned files

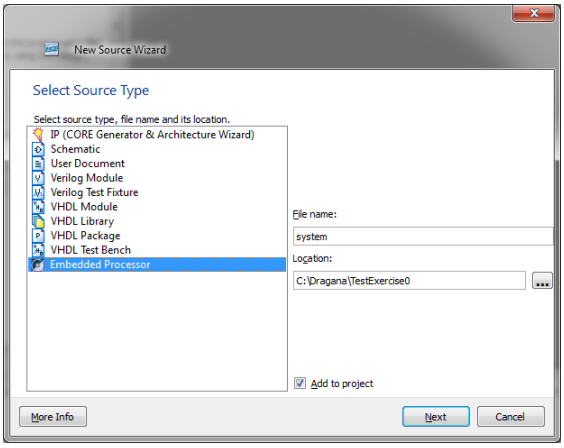

Figure 2.46: Adding a new source file corresponding to an embedded processor

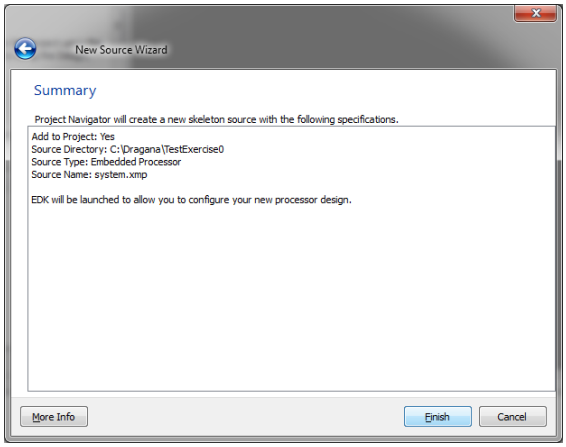

Figure 2.47: A summary of the newly created project

that 'EDK will be launched to allow you to configure your new processor design'. Click **Finish**.

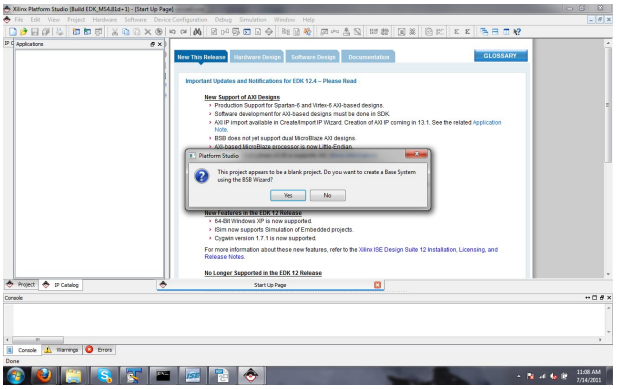

Figure 2.48: Switching from ISE Project Navigator to XPS

After a few moments, a Xilinx Platform Studio opens and a message appears as shown in Figure 2.48 where you are asked if you would like to use a Base System Builder (BSB) wizard to create your system. Click **Yes**.

Further, the wizard will ask you what type of bus you would like to choose for the connections within your embedded system. Choose the PLB as shown in Figure 2.49 and click **OK**. In the next window opened by the wizard choose the option **I would like to create a new design** and click **Next**, see Figure 2.50.

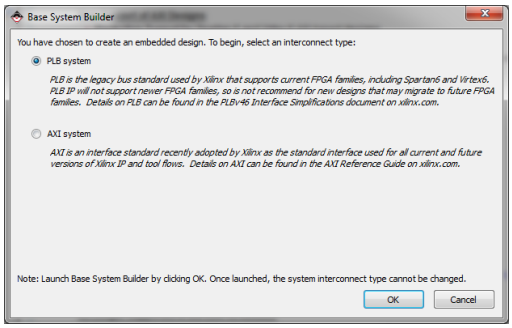

Figure 2.49: Choosing the type of the bus for the system: choose Processor Local Bus, PLB

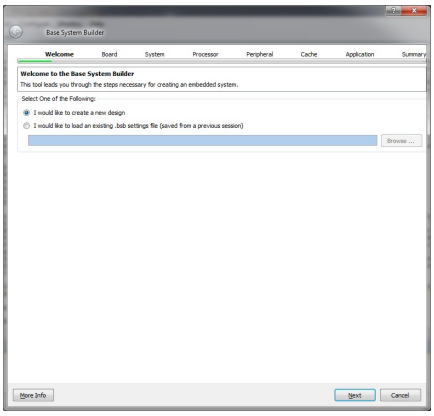

Figure 2.50: Creating a new design

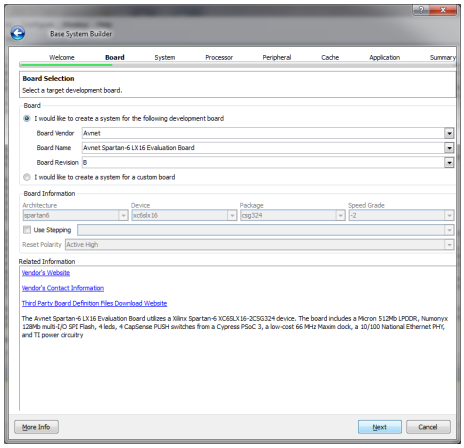

Figure 2.51: Choosing the development board for the hardware platform: choose Avnet board Avnet Spartan–6 LX16 Evaluation Board

Now the wizard asks you to specify for which board you will develop the embedded system. Make a choice as shown in Figure 2.51: for the **Board Vendor** choose Avnet, for the **Board Name** choose Avnet Spartan–6 LX16 Evaluation Board. Click **Next**.

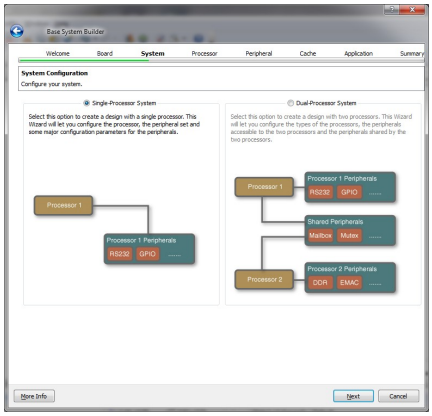

Figure 2.52: Choosing the number of processors for the system

Then, you are asked to define system configuration with respect to the number of processors. For the purpose of providing a communication for your design, one processor will suffice. Therefore, choose the option **Single–Processor System** as shown in Figure 2.52 and click **Next**.

Now you are asked to define some parameters for the processor. Make the choice as shown in Figure 2.53: MicroBlaze for the **Processor Type**, keep the system clock frequency as offered i.e. equal to the reference clock frequency and

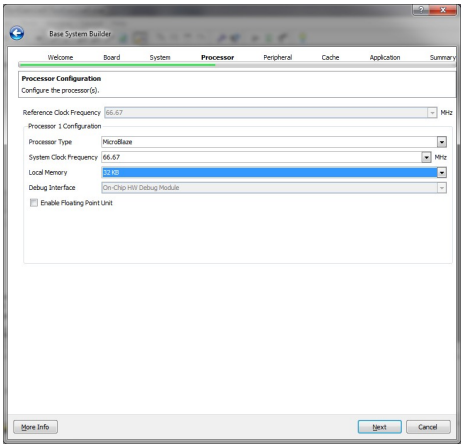

Figure 2.53: Defining processor parameters

for the **Local Memory** choose the amount of 32KB. Click **Next**.

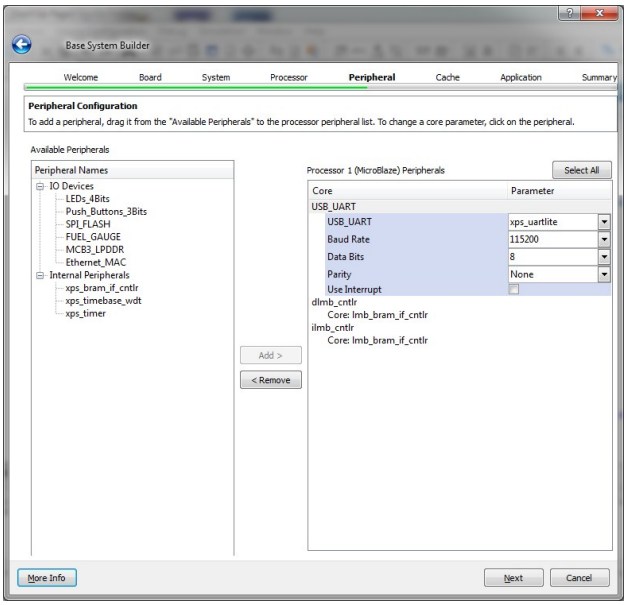

Figure 2.54: Choosing peripherals for the hardware platform: mark that the parameters for the UART peripheral are set so that they comply with the communication parameters for the Avnet board

Now that the processor is specified, you are asked to choose peripherals. Make the choice as shown in Figure 2.54, removing all the components apart from the component for the communication with UART and the components for

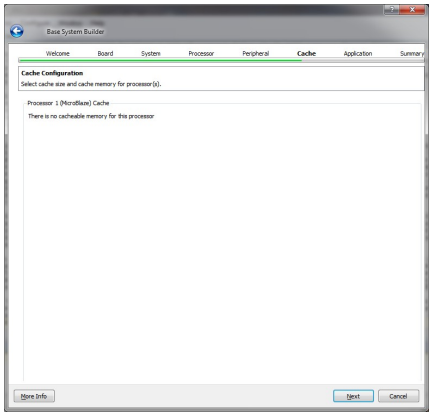

Figure 2.55: A system with no cache memory

data and instruction transfer between the MicroBlaze processor and Processor Local Bus (PLB) interconnections. Click **Next**.

In the next window just click **Next** as there is no cache memory, see Figure 2.55.

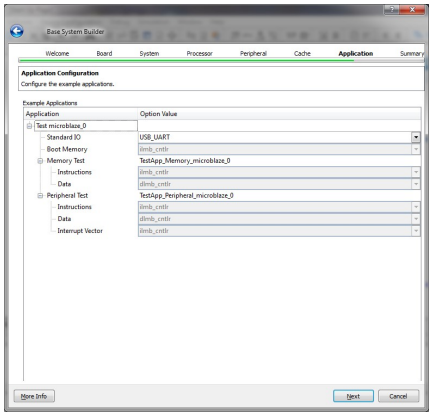

Figure 2.56: Applications for the platform to be created by the wizard

The Application window opens as shown in Figure 2.56. Mark that for the application Test_microblaze_0 as a standard I/O USB_UART is chosen. This corresponds to the peripheral you have chosen in one of the previous windows which will realise the needed communication for our design. Click **Next**.

A summary of your design appears as in Figure 2.57. Pay attention to the location of the files created in XPS pertaining to your system: they are placed in the folder **system** within the folder corresponding to the project opened in ISE Project Navigator environment which invoked the XPS. Also note the file **system.ucf** in the folder **data** within the **system** folder. This file contains the

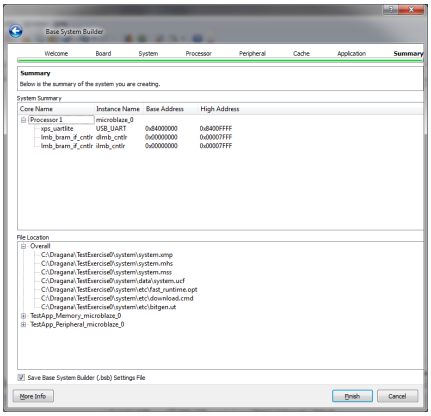

Figure 2.57: A summary of the hardware platform to be created

constraints for your system design which will be later used for the generation of the programming file. Click **Finish**.

Now you can view the system design created by Base System Builder wizard according to your specification see Figure 2.58. In the **Project** pane to the left, there are three tabs – the **Project** tab which contains information on the files associated with the project and basic project options, the **Applications** pane with the associated software applications (for the time being there are just those generated by the wizard) and the **IP Catalog** pane. It contains IP cores which can be incorporated into your system. For the time being there are only those pre–built but soon you will add one of your own – the design created in the previous section.

In the window to the right of XPS, there are again several tabs. Figure 2.58 shows one of them, **System Assembly View**. Go through the tabs and pay attention to which components are present in your system. If you are interested in learning more about XPS, look into supporting documentation like [3]. Figure 2.59 shows a block diagram of the created system.

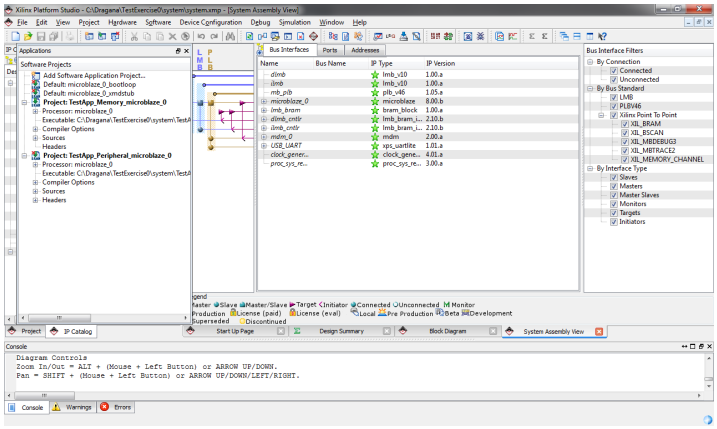

Figure 2.58: System view of the created design for the hardware platform

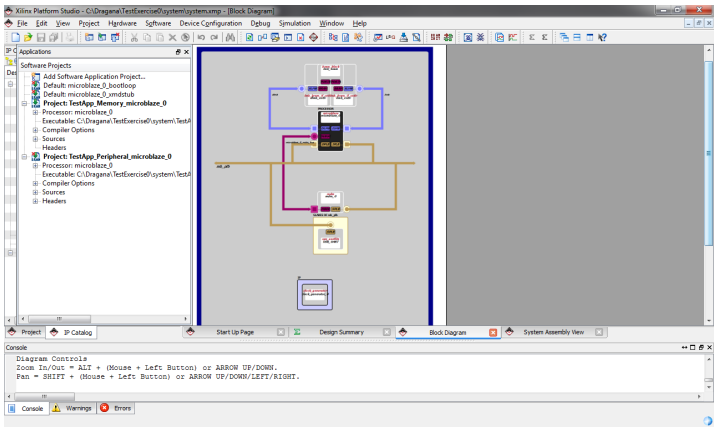

Figure 2.59: A block diagram of the created system

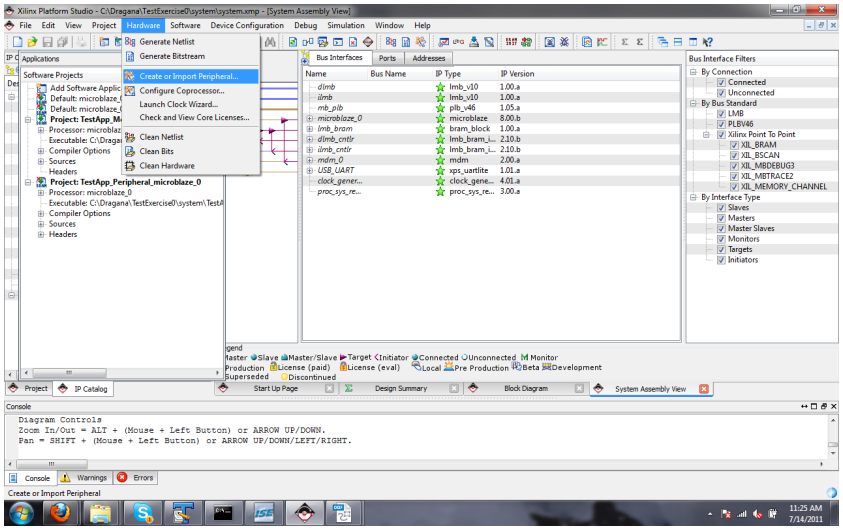

Figure 2.60: Invoking a **Create and Import Peripheral wizard** from the XPS menu

Create and Import Peripheral

You will add the designed incrementer (see section 2.4.1) as a user peripheral core to the hardware platform based on a MicroBlaze soft processor and a PLB. However, for this purpose a design from section 2.4.1 needs to be extended with the module which will enable the incrementer to communicate with the rest of the platform hardware over the PLB. This module will realise the corresponding communication protocol.

We shall use again a wizard to help us create a peripheral core. Figure 2.60 shows how you can start a Create and Import Peripheral (CIP) wizard by choosing the corresponding menu option. A welcome window appears as in Figure 2.61. Click **Next**.

In the window which opens choose the option **Create templates for a new peripheral**, see Figure 2.62, as you will need to create a new peripheral module which implements the logic of the incrementer created in section 2.4.1 but also the communication protocol with the PLB system bus. Click **Next**.

For the location where the files pertaining to your peripheral will be saved choose **To an XPS project** as shown in Figure 2.63. As indicated at the bottom of the screen, the files will be saved in the subfolder **pcores** within your project tree. Click **Next**.

Choose the name for the peripheral, **incrementer** as for the example in Figure 2.64.The version will be automatically set for you by the wizard. Click **Next**.

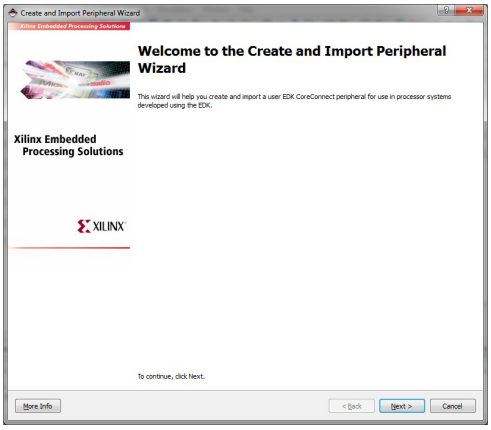

Figure 2.61: CIP wizard welcome window

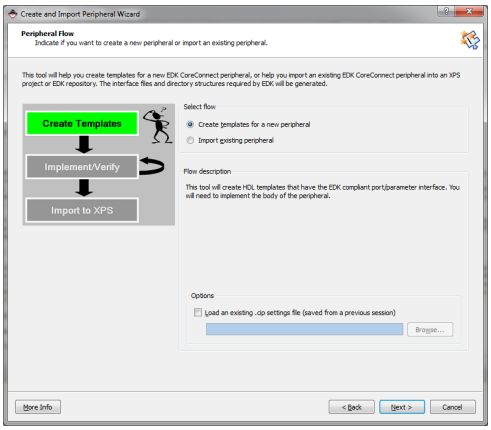

Figure 2.62: Choosing the option Create new peripheral

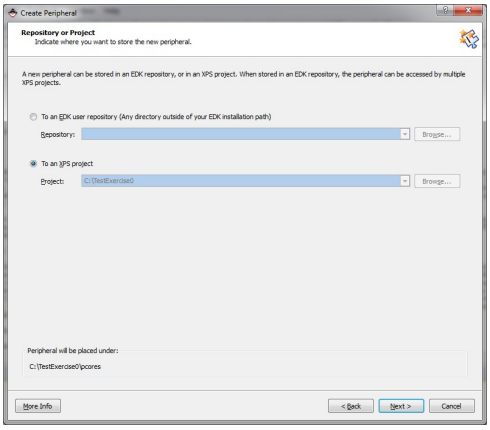

Figure 2.63: Save the new peripheral within your XPS project tree

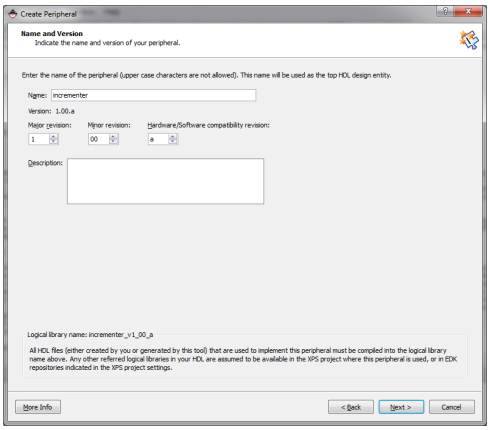

Figure 2.64: Assigning the name and version for the peripheral

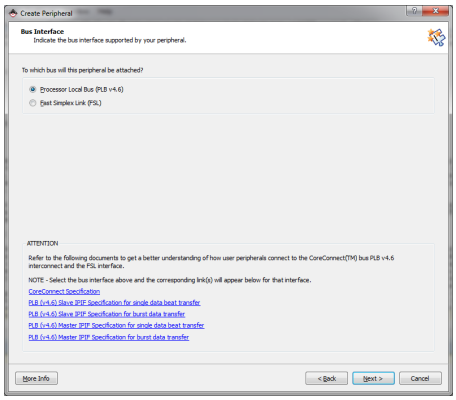

Figure 2.65: The choice for the PLB interface for the peripheral

For the interface for the new peripheral choose PLB as in Figure 2.65 and click **Next**.

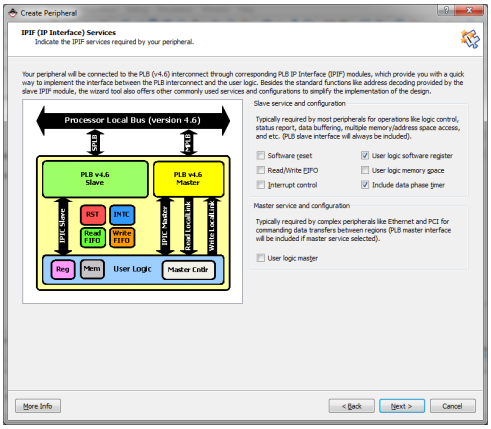

Figure 2.66: Choosing interface resources for the peripheral

In the window which opens you may choose interface resources for the peripheral. For this simple example of an incrementer, make the selection as shown in Figure 2.66 and click **Next**.

Your peripheral was chosen to interface the system bus as a slave. In embedded systems, the communication is realised via buses. In our system, the system bus was chosen to be a PLB type. All components connected to the system bus, the processor and peripherals alike, need to obey certain communication protocol during the data transfer via bus lines. In our system a Master/Slave mode was chosen in which one component, the MicroBlaze processor in our case, is a Master of all the data transfer via PLB, acting as an initiator of the transfers.

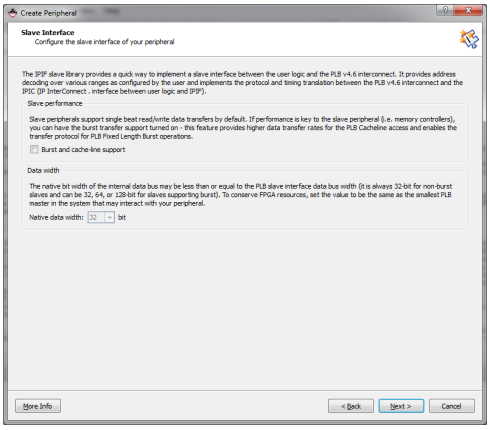

Figure 2.67: PLB Slave configuration for the peripheral

Peripherals are assigned the role of slaves. So does the incrementer acts as a slave within the communication protocol. For the configuration of the slave interface which appears, see Figure 2.67, choose the offered solution where no bursts or use of cache are allowed for the transfer and click **Next**.

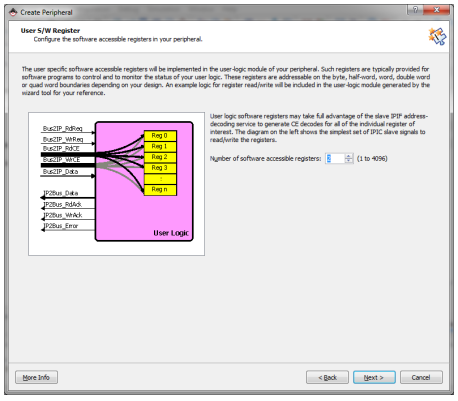

Figure 2.68: Choosing the number of software accessible registers: 2 in our case

In this window you can choose the number of registers which can be accessed through the software i.e. which can be read and written by your program. We shall make a choice of 2 because we plan to use one register for writing the data for the incrementer and another for reading the data from the incrementer output. Click **Next**.

The window shown in Figure 2.69 leaves you the option of choosing which interconnect lines to leave between the new peripheral and the rest of the system. By highlighting a line in the middle pane, the corresponding description appears

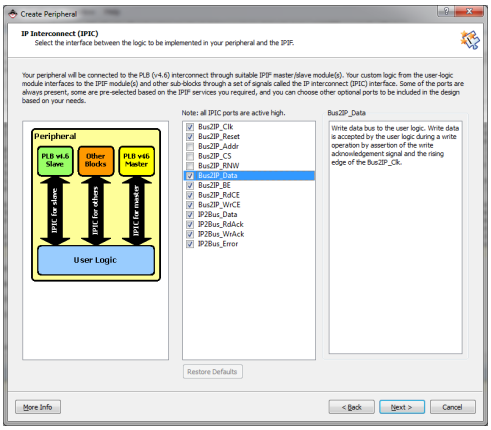

Figure 2.69: Interconnect lines between the new peripheral and the rest of the system

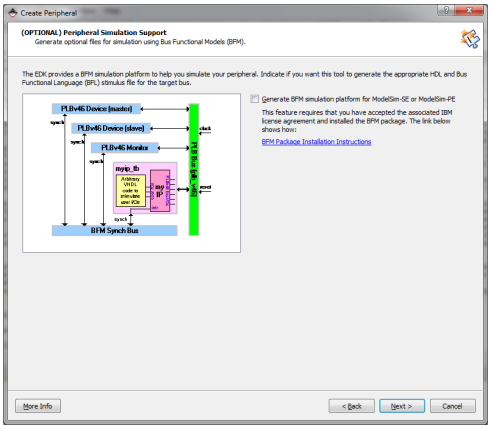

Figure 2.70: No simulation files to be added in the platform

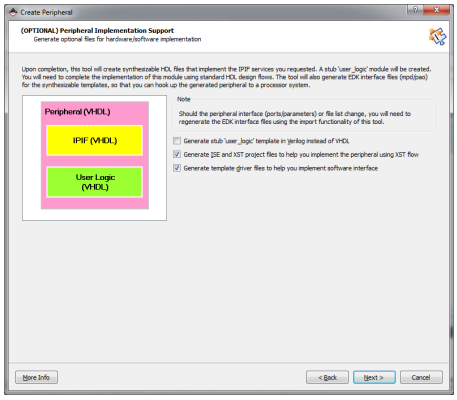

Figure 2.71: Asking wizard to create support ISE and XST files for the peripheral and examples for the supporting drivers

in the right pane. In Figure 2.69, you are shown an example of the description for the data lines from the PLB bus to the peripheral. Also, mark the schematic view in the left pane. The logic you implemented for the incrementer in section 2.4.1 will be placed in the block **User Logic**. Choose the suggested interface lines and click **Next**. In the next window, see Figure 2.70, click **Next** without ticking the offered possibility to generate test bench files. This is because all the simulations for your logical design will be done in ModelSim beforehand.

In the next window, see Figure 2.71, tick the two options which will help you create your peripheral – for the wizard to generate ISE and XST project files and driver files for the software interface. The former ones will help you implement your design while the latter ones will provide you with the skeleton and example code for accessing and making use of your peripheral. Click **Next**.

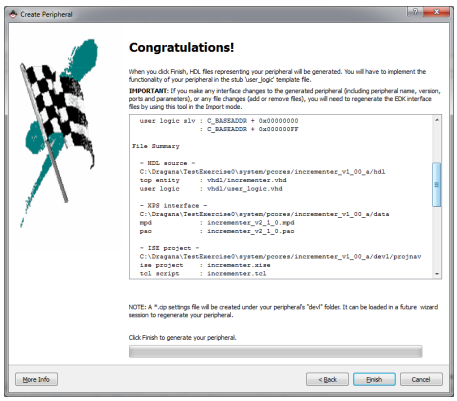

Figure 2.72: A summary of the support files for the created peripheral

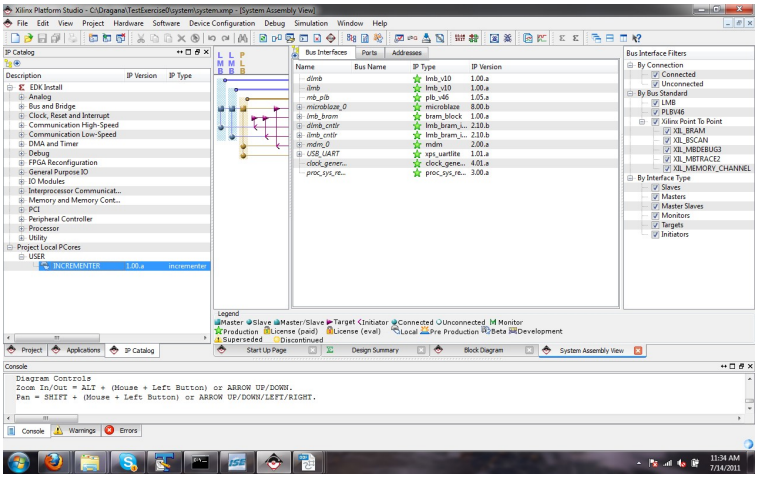

Figure 2.73: The new user peripheral core added among the available IP cores

A window with the information about the new peripheral opens as in Figure 2.72. You are advised to scroll through the central pane and revise once more what files will be created and where they will be placed. Click **Finish**.

The newly created peripheral is shown among other IP cores under **Project Local Cores** subtree, see Figure 2.73.

The user logic design, i.e. the logic of your incrementer, will be added within ISE Project Navigator for the project created by CIP wizard. As specified, the accompanying files are placed in one of the subfolders within the project tree as shown in Figure 2.74.

Close the XPS environment and open this project in ISE Project Navigator. Figure 2.75 shows this project opened. Two main .vhd files can be seen in the **Project** pane: **plbv46_ slave_ single.vhd** which implements the design needed for the communication with the PLB and **user_logic.vhd** which will be updated with the design of the incrementer from the section 2.4.1. Open this file in the right pane and scroll through its contents. You will see that CIP wizard has created most of the interface and PLB–related logic for you. It has also marked the sections which you should not change and the sections in which you are free to add your own design. Scroll to the section for the declaration of signals, variables and components which will be used in the description of the entity's behaviour and add the following code, as shown in Figure 2.76:

```
component toplevel
      port ( clk : in  STD_LOGIC;
         reset : in  STD_LOGIC;
         bus_in : in  STD_LOGIC_VECTOR (31 downto 0);
         bus_out : out  STD_LOGIC_VECTOR (31 downto 0));
end component;
```

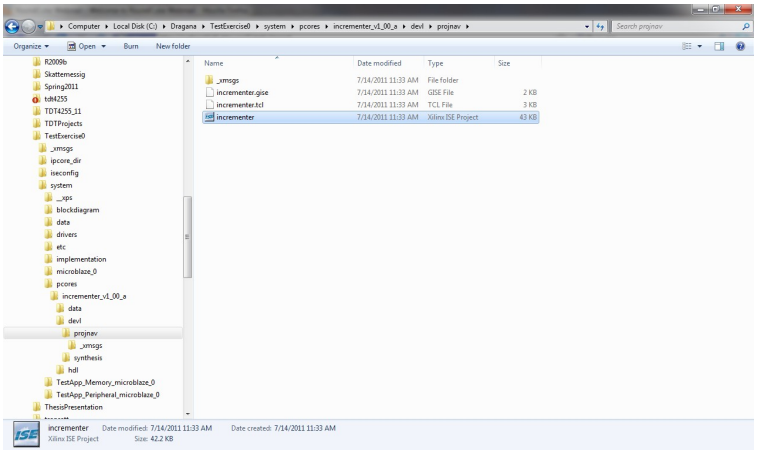

Figure 2.74: The placement of the files which can be used for further development of the peripheral logic within ISE and XST

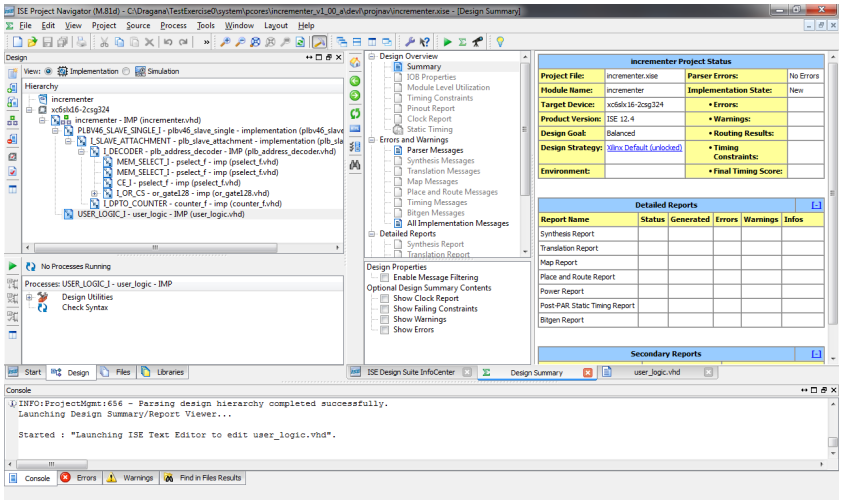

Figure 2.75: ISE project for the peripheral, located at <ISE_system_project>\system\pcores\<your_pcore>\devl\projnav\

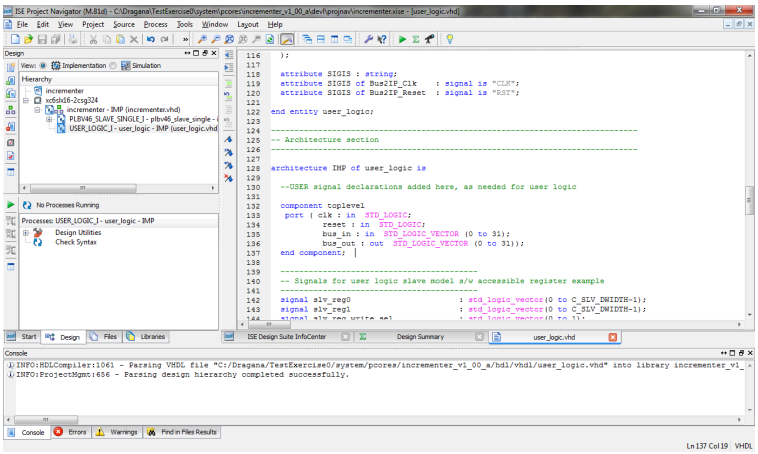

Figure 2.76: User_logic.vhd file and the section for the implementation of the incrementer architecture: designed incrementer from section 2.4.1 is included as a component

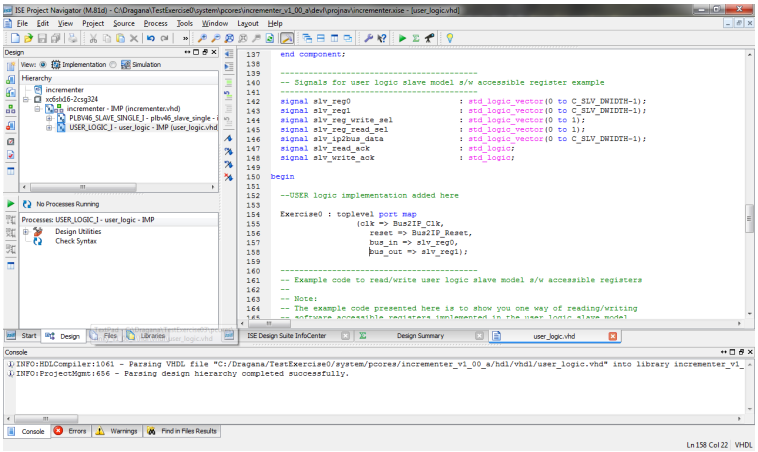

Figure 2.77: User_logic.vhd file and the section for the implementation of the incrementer architecture: instantiation of the incrementer component

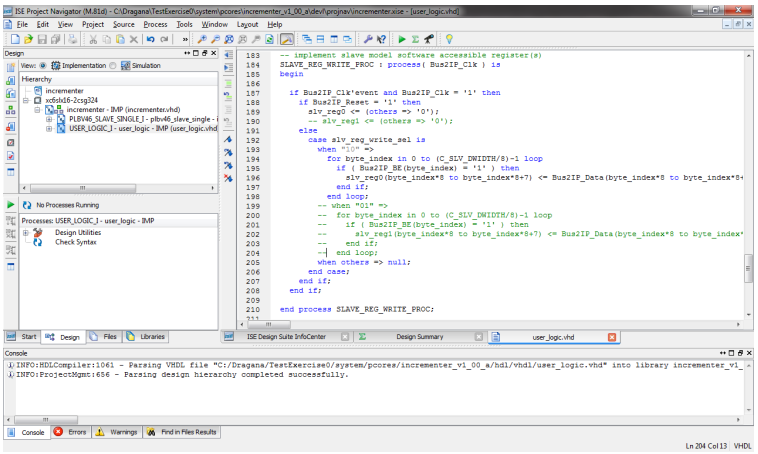

Figure 2.78: User_logic.vhd file and the section for the implementation of the process 'SLAVE_REG_WRITE_PROC'

You will use the incrementer design from section 2.4.1 as a component within a larger system so the incrementer component needs to be instantiated. Scroll through the **user_logic.vhd** file to the section of the architecture description and add the following code below the wizard–generated line –**USER logic implementation added here**, as shown in Figure 2.77:

```
Exercise0 : toplevel port map
              (clk => Bus2IP_Clk,
               reset => Bus2IP_Reset,
               bus_in => slv_reg0,
               bus_out => slv_reg1);
```

It instantiates the incrementer component with the signals within the architecture of the user_logic block within the peripheral. Scroll further down to the process which describes the write operation of the data from the PLB to the software accessible registers. It is our intention to use register 0 for writing and register 1 for reading from the peripheral core. This can be seen from the signal assignments in the port map list for the `bus_in` and `bus_out`. Therefore, comment or delete the lines related to register 1 when writing the data for the incrementer with the data from PLB, see Figure 2.78. Your code for the wizard–generated process 'SLAVE_REG_WRITE_PROC' should look like this:

```
SLAVE_REG_WRITE_PROC : process( Bus2IP_Clk ) is
begin

  if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
    if Bus2IP_Reset = '1' then
      slv_reg0 <= (others => '0');
      -- slv_reg1 <= (others => '0');
```

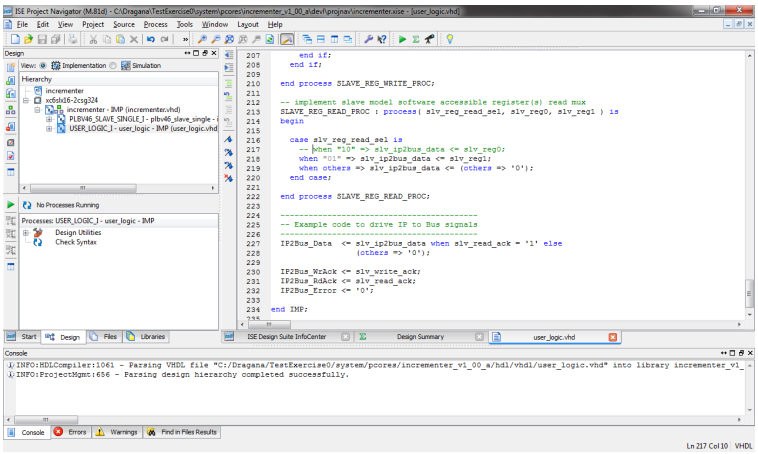

Figure 2.79: User_logic.vhd file and the section for the implementation of the process 'SLAVE_REG_READ_PROC'

```
    else
      case slv_reg_write_sel is
        when "10" =>
          for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
            if ( Bus2IP_BE(byte_index) = '1' ) then
              slv_reg0(byte_index*8 to byte_index*8+7) <=
                    Bus2IP_Data(byte_index*8 to byte_index*8+7);
            end if;
          end loop;
        -- when "01" =>
        --   for byte_index in 0 to (C_SLV_DWIDTH/8)-1 loop
        --     if ( Bus2IP_BE(byte_index) = '1' ) then
        --       slv_reg1(byte_index*8 to byte_index*8+7) <=
        --     Bus2IP_Data(byte_index*8 to byte_index*8+7);
        --     end if;
        --   end loop;
        when others => null;
      end case;
    end if;
  end if;

end process SLAVE_REG_WRITE_PROC;
```

Analogously, for the process 'SLAVE_REG_READ_PROC', comment or delete the line related to reading of the register 1 as shown in Figure 2.79. The process 'SLAVE_REG_READ_PROC' in your code should look like this:

```
SLAVE_REG_READ_PROC : process( slv_reg_read_sel, slv_reg0, slv_reg1 ) is
begin

  case slv_reg_read_sel is
    -- when "10" => slv_ip2bus_data <= slv_reg0;
```

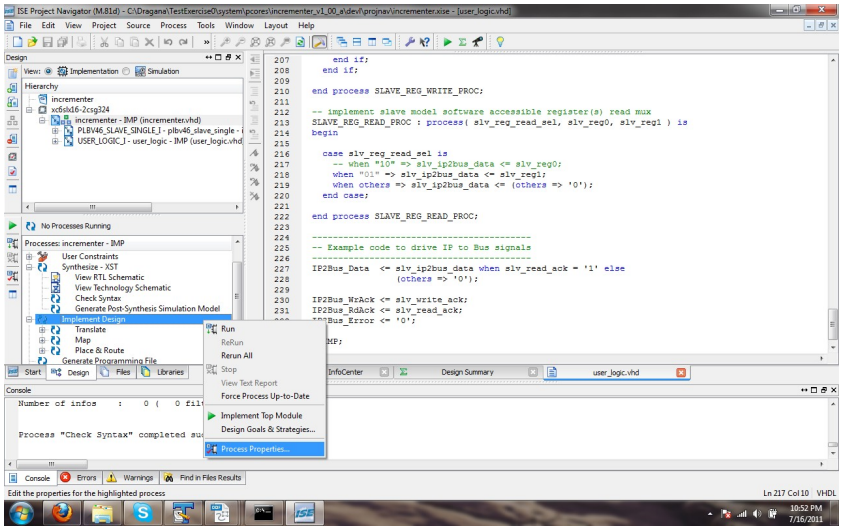

Figure 2.80: Adding the path for the **Implement Design** process

```
    when "01" => slv_ip2bus_data <= slv_reg1;
    when others => slv_ip2bus_data <= (others => '0');
  end case;

end process SLAVE_REG_READ_PROC;
```

Your peripheral core now contains the logic implemented in another project, the project you made in section 2.4.1. For further synthesis and implementation within a larger system, it is necessary to provide the path for the location of this file. Highlight the name of the top level in the upper pane of the **Design** tab and right–click on the pop–up menu item **Process Properties** which opens when **Implement Design** is highlighted in the **Processes** pane, see Figure 2.80. Choose the location of your project files as shown in Figure 2.81. Click **Apply** and then **OK**. Check the syntax and exit ISE Project Navigator environment.

You will open XPS to add the updated peripheral in your hardware platform. Choose the menu option **Hardware** → **Create and Import Peripheral** as in Figures 2.60, 2.61. In the window which opens after the welcome window, choose **Import existing peripheral** option, see Figure 2.82.

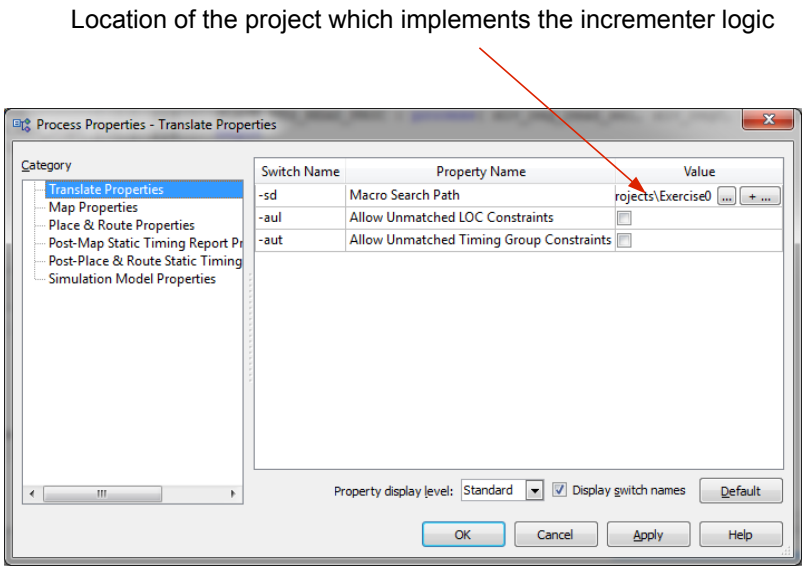

Figure 2.81: Adding the path of the project with the .vhd file which contains the description of the incrementer design

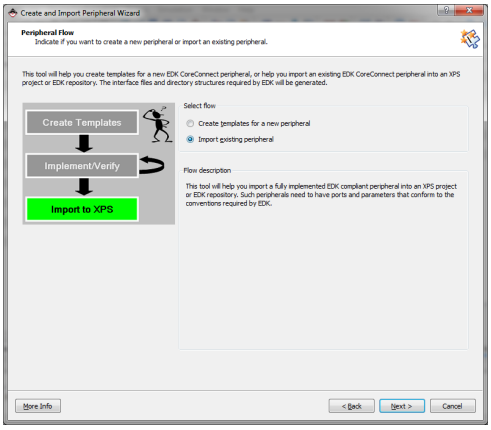

Figure 2.82: Choosing to import the peripheral

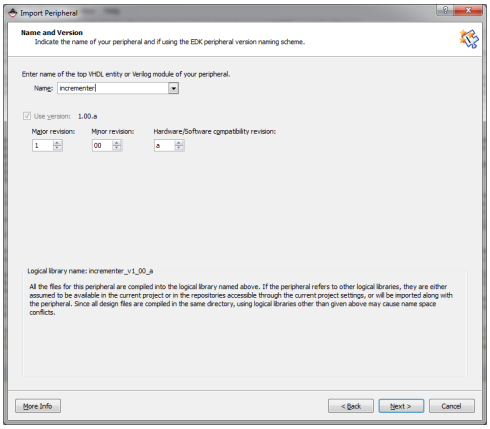

Figure 2.83: Assigning the name and version to the peripheral to be imported

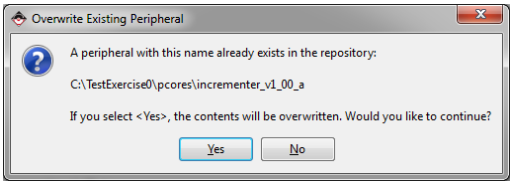

Figure 2.84: Confirmation that the peripheral to be imported should overwrite the contents of previous files

As in Figure 2.63 choose the XPS project as the place where to save the corresponding files. Choose the **incrementer** for the name of the toplevel for the peripheral and accept the offered version, see Figure 2.83. Click **Next**. For the notification that a peripheral with such name already exists, click **Yes** that you want to overwrite its contents as shown in Figure 2.84.

For the type of the source files choose the first option – **HDL source files** and click **Next**, see Figure 2.85.

Now you are asked to define the source of your peripheral–related .vhd files. Because the CIP wizard has already created Peripheral Analysis Order file and placed it in the **data** folder below the folder corresponding to the new peripheral in the project tree, you are advised to use this file as a source of your design files. As Figure 2.86 shows, locate this file through the **Browse** button and

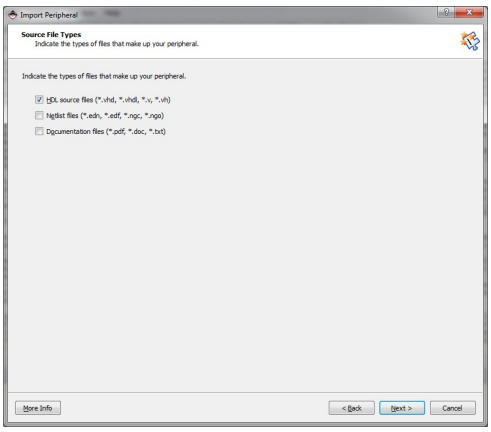

Figure 2.85: Opting for the source files to be provided in the HDL form, VHDL in our case

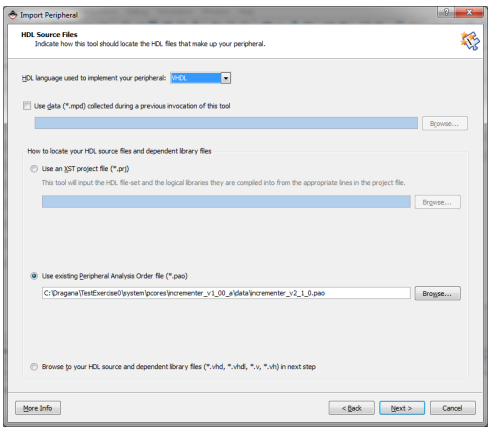

Figure 2.86: Providing the Peripheral Analysis Order file as a source for the import of the peripheral

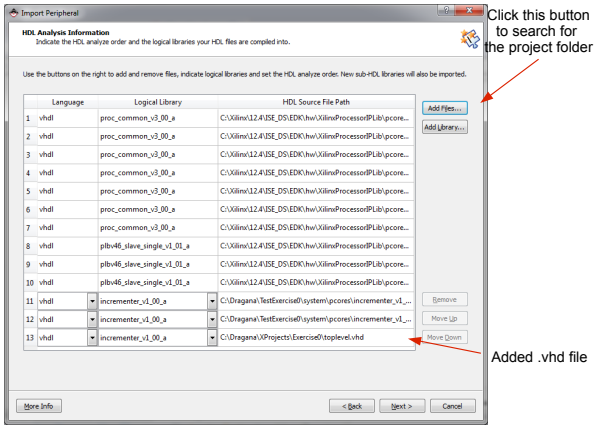

Figure 2.87: A list of VHDL–files which will be included for the imported peripheral

click **Next**.

In the window which opens, see Figure 2.87, a list of .vhd files is shown which will be included into the imported peripheral. Pay attention to the two bottom lines. They refer to .vhd files you have modified in ISE Project Navigator so as to include the logic of the incrementer. Click **Next**.

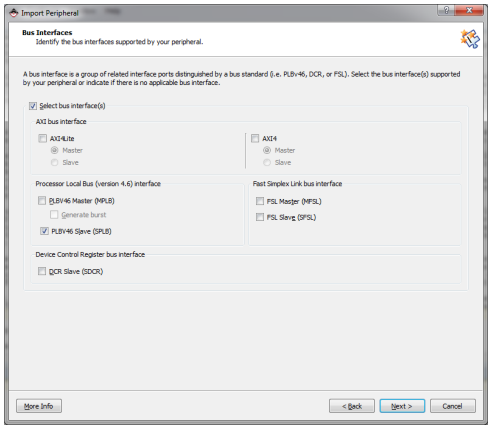

Figure 2.88: Choosing the peripheral communication mode as a slave to a system PLB bus

For the interface between the new peripheral and the system bus choose PLBv46 Slave as shown in Figure 2.88. Click **Next**.

Scroll through the list of ports which opens in the next window, as shown in Figure 2.89. Notice that the CIP wizard has created the signals for the interface

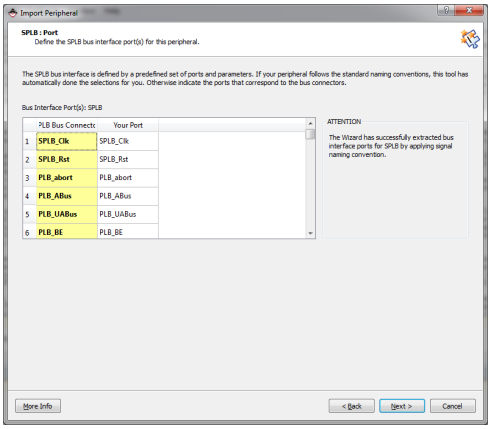

Figure 2.89: Peripheral ports

between the peripheral and the rest of the system obeying the signal naming convention so that it is easy to determine the connections between the two. Click **Next**.

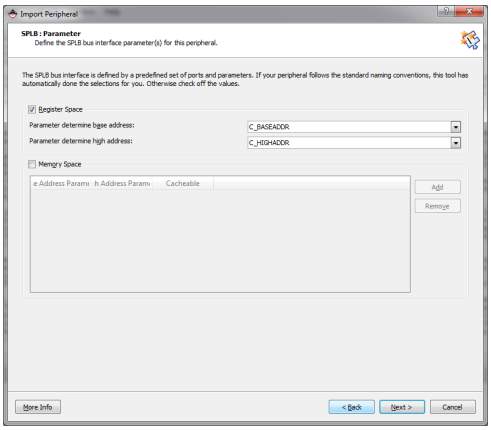

Figure 2.90: Assigned parameters for the peripheral core

Click **Next** for the offered assigned parameters, Figure 2.90 and for the interrupts – there were none included, see Figure 2.91.

Also, for the two windows which follow regarding the user defined parameters and ports, accept offered choice and click **Next**, see Figures 2.92 and 2.93.

A peripheral summary appears as shown in Figure 2.94. Click **Finish**.

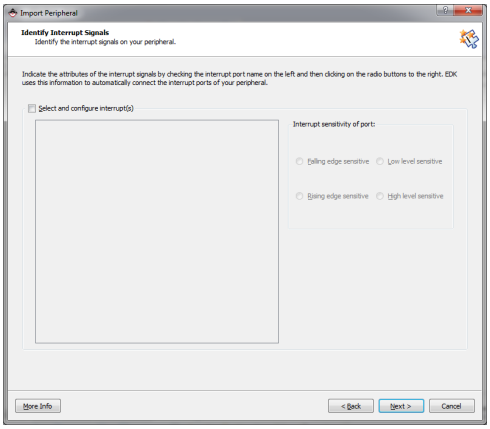

Figure 2.91: No interrupts included

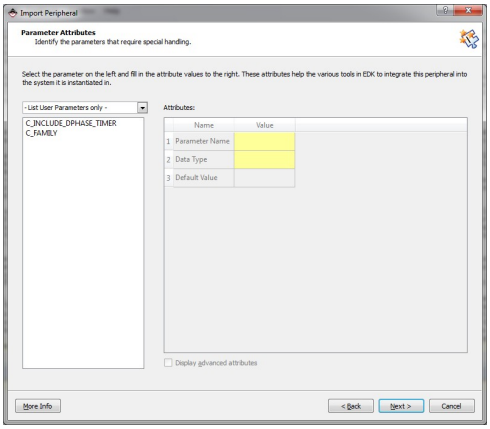

Figure 2.92: User defined parameters

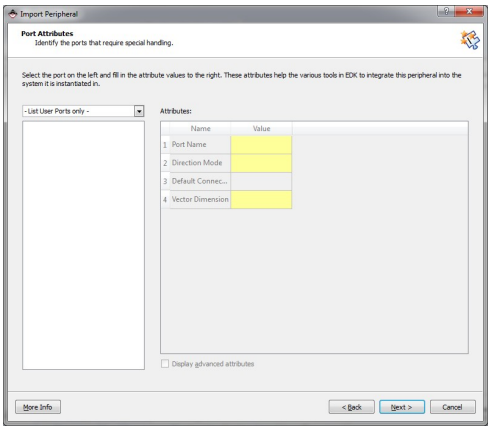

Figure 2.93: User defined ports

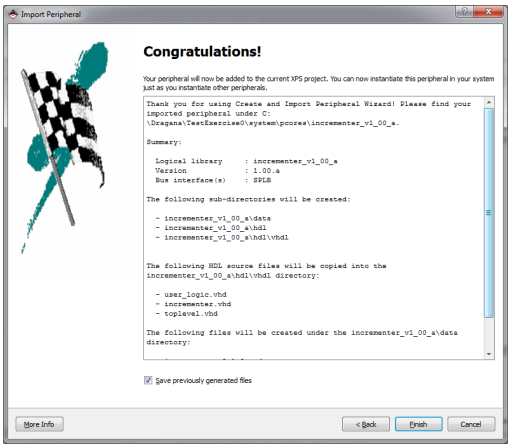

Figure 2.94: A summary for the imported peripheral

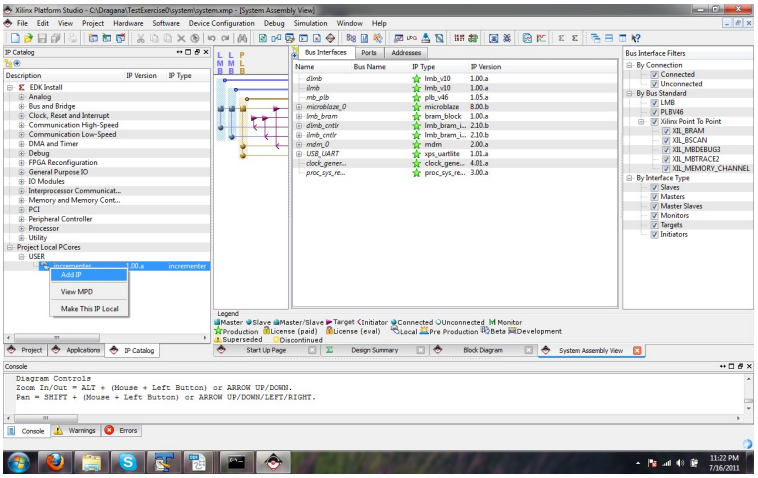

Figure 2.95: Adding peripheral to the hardware platform

After the peripheral is imported, it needs to be added to the hardware platform. Right–click on the peripheral on the **Add IP** item in the pop–up menu to add the peripheral in the System Assembly View in the central pane, see Figure 2.95.

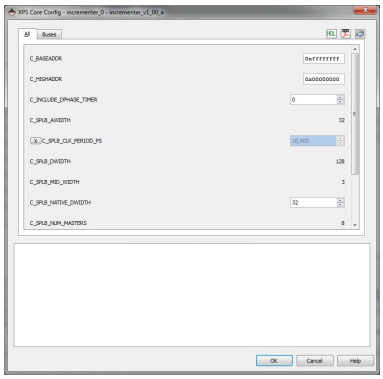

Figure 2.96: The properties of the new peripheral within the hardware platform

In the window which opens you may view the properties of the new peripheral within the hardware platform, see Figure 2.96. Click **OK**.

Now the instance of the new peripheral appears in the System Assembly pane but it is still not connected to any of the buses within your hardware platform. In order to connect the Slave PLB interface of the incrementer peripheral, open

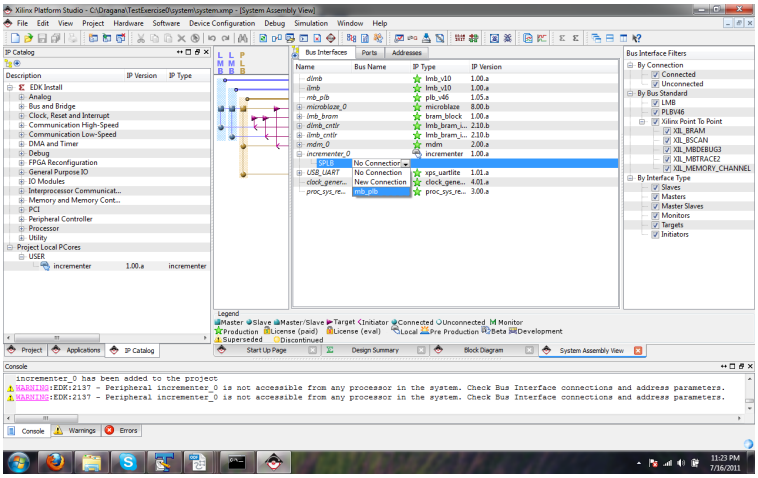

Figure 2.97: Connecting the added incrementer peripheral to the system PLB bus

the drop–down list by the SPLB interface of the *incrementer_ 0* peripheral and choose *mb_plb* as shown in Figure 2.97. This is the MicroBlaze mastered PLB system bus.

Now your peripheral is connected to it, inspect the schematic of the bus connections to the left. Inspect the **Ports** and **Addresses** tabs of the System Assembly View. In the **Ports** tab for the **incrementer_ 0**, you may notice that its SPLB port is connected to the mb_ plb bus as previously defined, see Figure 2.98.

In the **Addresses** tab, under the **Unmapped Addresses**, assign 64K address space to the new peripheral, see Figure 2.99, and then click on the **Generate Addresses** button in the top right corner.

As Figure 2.100 shows, the newly added peripheral is assigned the base address 0x84418000.

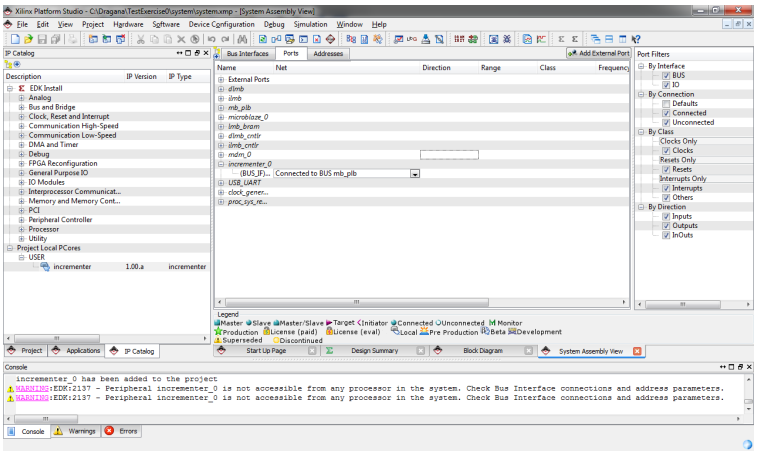

Figure 2.98: Peripheral SPLB port

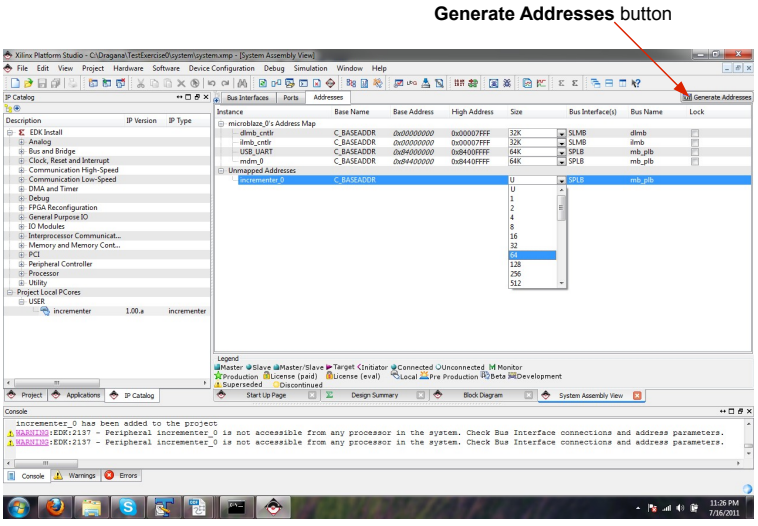

Figure 2.99: Assigning the address space to the peripheral

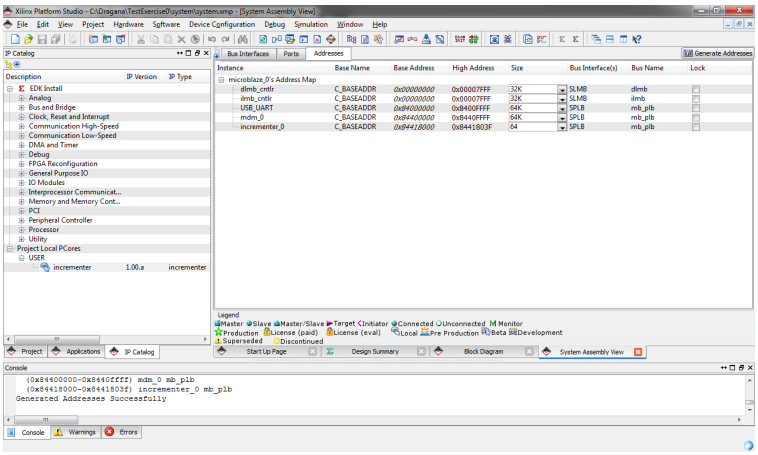

Figure 2.100: System peripherals after generation of the corresponding addresses

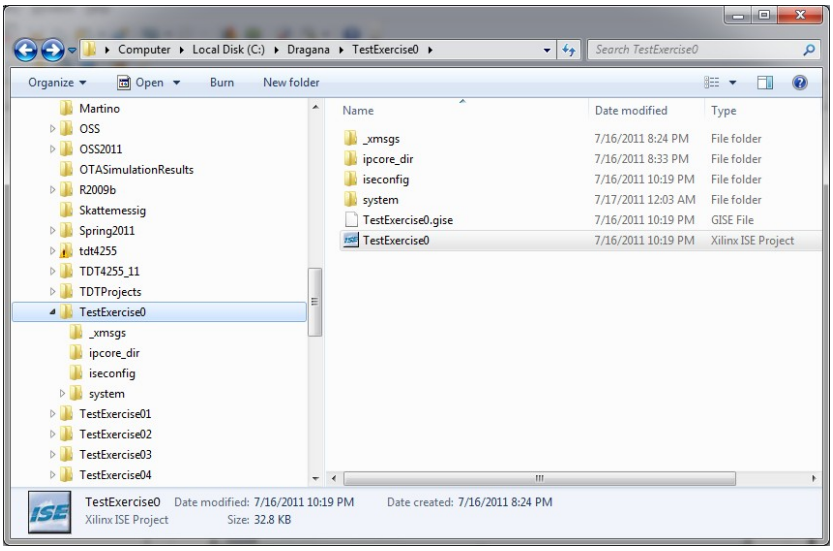

Figure 2.101: The location of the ISE project for the embedded processor system

Programming FPGA

Now we need to configure the FPGA – Spartan–6 chip on the Avnet development board – according to the system design. In other words, we need to program the FPGA to perform an incrementer functionality. In addition, we shall make a short software program which will run on FPGA. It will enable the user to send a number to the FPGA and read the response from the incrementer. By inspecting these values, the operation of the incrementer can be verified.

The FPGA is programmed by downloading a configuration .bit file. In section 2.4.1, we have seen how a .bit file is generated within ISE Project Navigator environment. In the same way we shall generate a programming .bit file for the system we have designed. In addition, as we intend to download the executable .elf file for the pertaining software as well, the generated .elf file needs to be merged with the configuration .bit file so as to produce a .bit file which is to be downloaded to the FPGA. The content of the .elf file will reside in the Spartan–6 BRAM memory. Therefore merging of the .bit file and the .elf file corresponds to the update of the .bit file so that it contains the data from the .elf and configures the BRAM with these data.

Let us now first generate a .bit file for the designed system. Open the Project Navigator for the system. It is the project you initially made for the embedded processor system. Figure 2.101 shows its location for our example. The only component is **system.xmp** which contains the information about the system design except the information on the constraints. Therefore, the constraints file needs to be added manually. When the design was created in the XPS

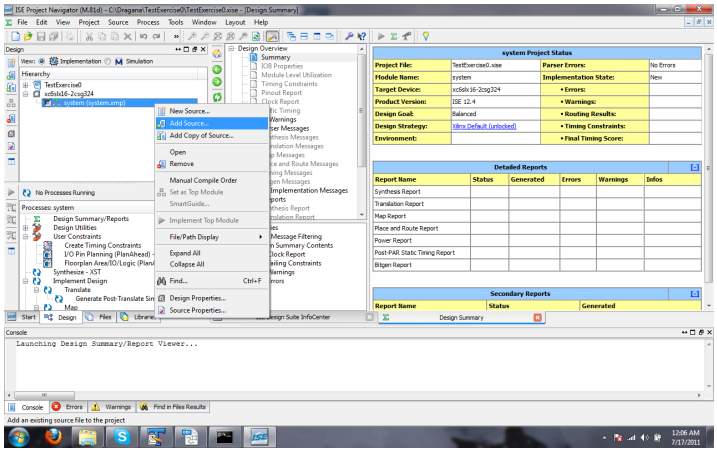

Figure 2.102: Adding a new source to the embedded system

environment, a constraints file was generated, as said, and now you will add this file to the system design in ISE Project Navigator so that the constraints are included as well.

Right–click the **system.xmp** source and click on the **Add Source** item in the pop–up menu which opens, see Figure 2.102.

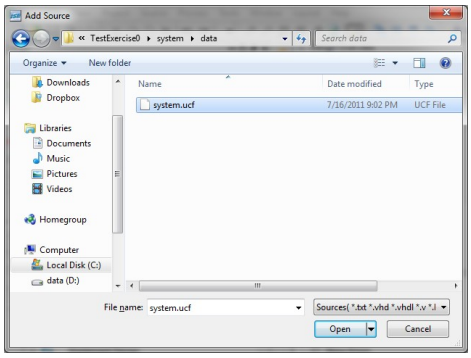

Figure 2.103: The location of the constraints file within the generated system platform

Locate the **system.ucf** file in the `<project_ name>\system\data\` folder as shown in Figure 2.103.

Now your project should look like in Figure 2.104. Highlight the top level source in the **Design** tab and click on the **Generate Programming File** in the **Processes** pane below. You will get some warnings at the **Translate**

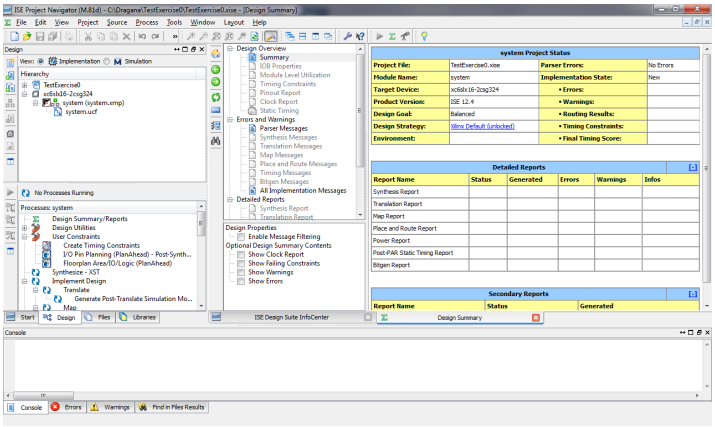

Figure 2.104: Constraints file added to the project

and **Place** & **Route** phases but ignore them as they refer to unrouted signals within the system which are not needed for our simple design. For the software development, we shall use XPS environment.

Open the project in XPS by double–clicking on the top level system(system.xmp) in the **Design** pane, see Figure 2.105. Click on the tab **Applications** on the left side.

Figure 2.106 shows its appearance and two test applications which were created by the wizard. On the **Software** menu click on the **Generate Libraries and BSPs** item, as shown in Figure 2.107. Click on the item **Assign Default Drivers**, see Figure 2.108. Now you will create a new application which will test the incrementer. Click on the **Add Software Application Project** as shown in Figure 2.109.

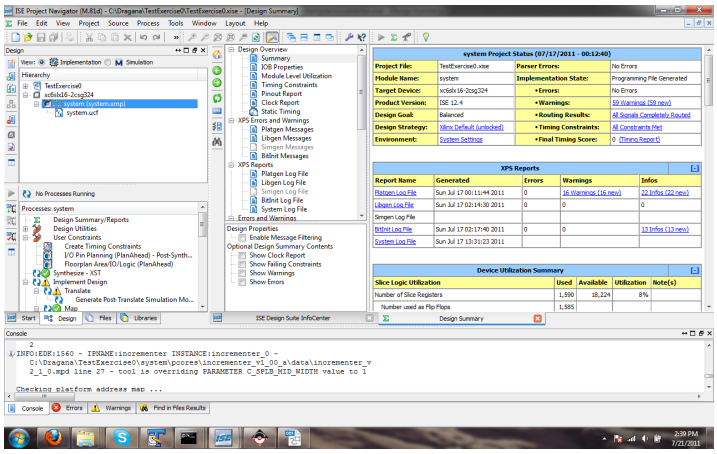

Figure 2.105: Opening the embedded platform in the XPS: double–click on system.xmp file

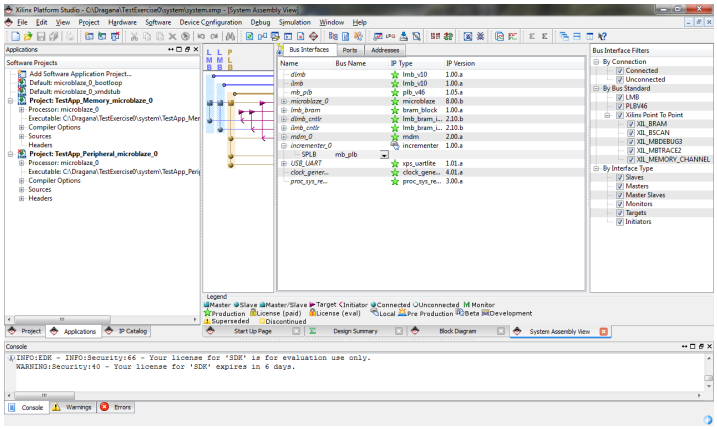

Figure 2.106: The system view in XPS: **Applications** tab shows test applications created by the wizard

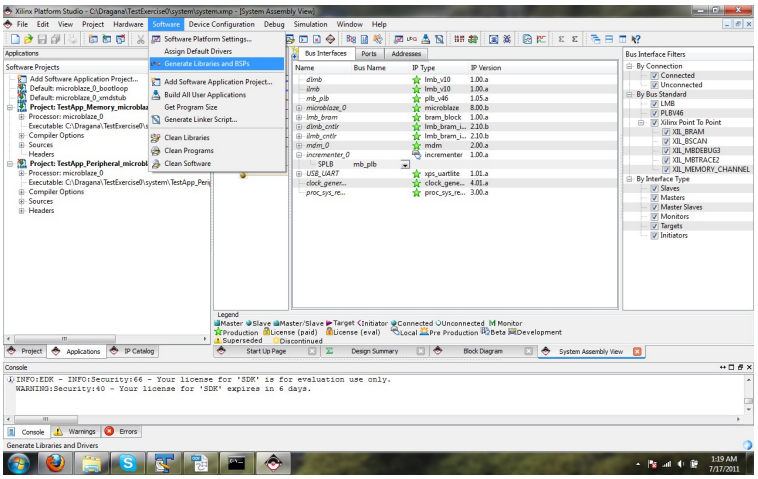

Figure 2.107: Generating libraries for the system applications

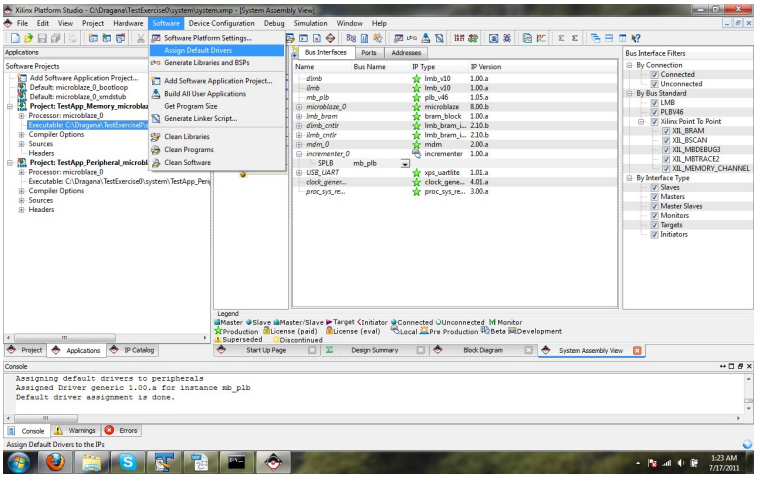

Figure 2.108: Assigning default drivers

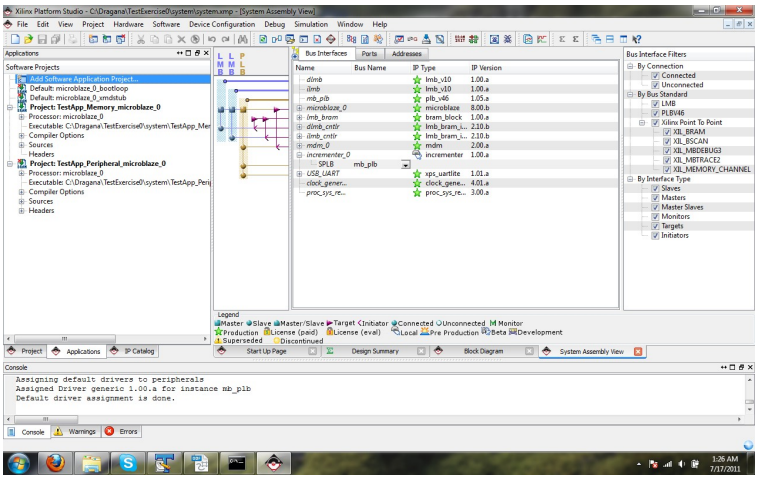

Figure 2.109: Adding a new software application

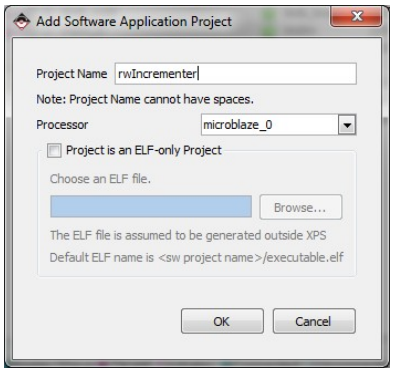

Figure 2.110: Choosing the name for the project

A window opens in which you are asked to provide the name for the project. In the example shown in Figure 2.110, a 'rwIncrementer' is chosen for the project's name. Click **OK**. Your application project is now added to the list of applications as Figure 2.111 shows.

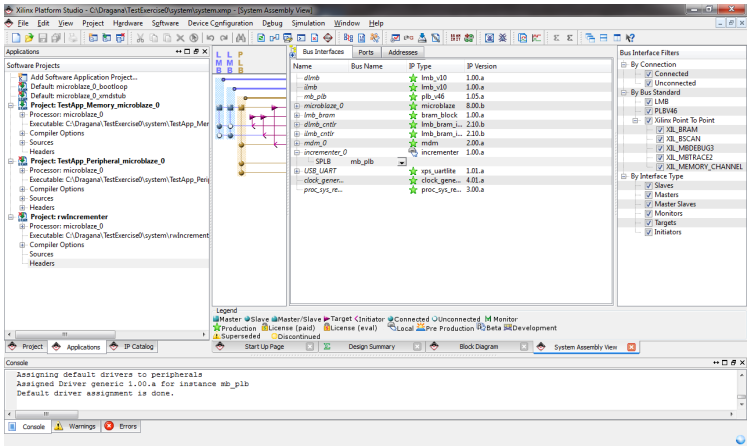

Figure 2.111: New application project

Before you continue with writing a program, make a folder within the project tree of folders where you will save the files related to your project. When you take a look at the folder `<project_name>\system`, you will notice that there is a folder for each of the two test applications generated by the wizard. Within the project folder, source files are kept in the **src** directory. Before you continue,

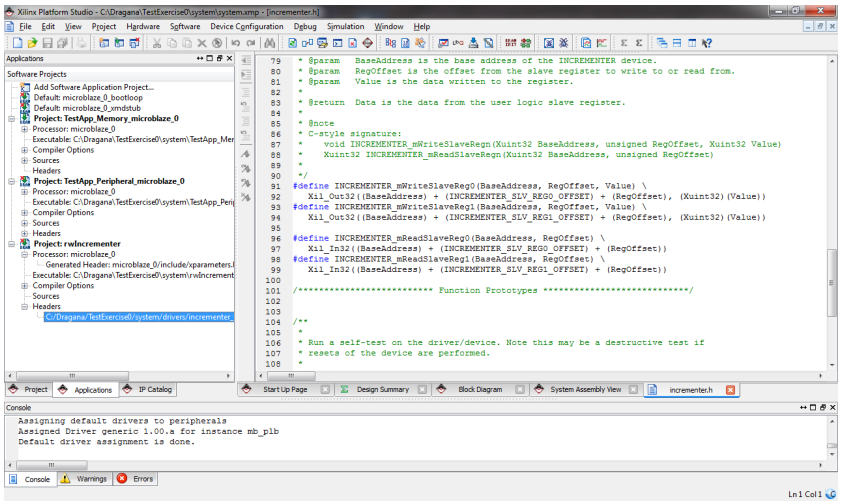

Figure 2.112: The header file added to the project

alongside the test projects folders, create a folder named after the project i.e. **rwIncrementer** with the subfolder **src**.

When you invoked the option **Generate Libraries and BSPs Scripts** from the menu item **Software**, among the other created files, there was a file with declarations and some definitions for the peripheral core you have created. These can be used for software access to the peripheral core. The file is named after the peripheral core and has the extension .h according to the C programming language convention for header files.

The generated **incrementer.h** file is at the location `<project_name>\-system\drivers\<peripheral_name_and_version>\data\source`. Open the file and inspect its contents.

Figure 2.112 shows one segment of it with the definition of the function for accessing the two software accessible registers. These functions will be used in our example program.

Now you will add the source code for your application. As shown in Figure 2.113, right–click on the node **Sources** in the project tree and choose the location and name for your .c file.

Figure 2.114 shows the **src** folder within your project folder and the name rwMain.c for your source file. Click **Save**.

Now your .c file is added to the project, see Figure 2.115 left pane. When you double–click the **rwMain.c**, the blank file will open in the main pane to the right. Add the code shown in Figure 2.115. For convenience, the listing is provided in sequel:

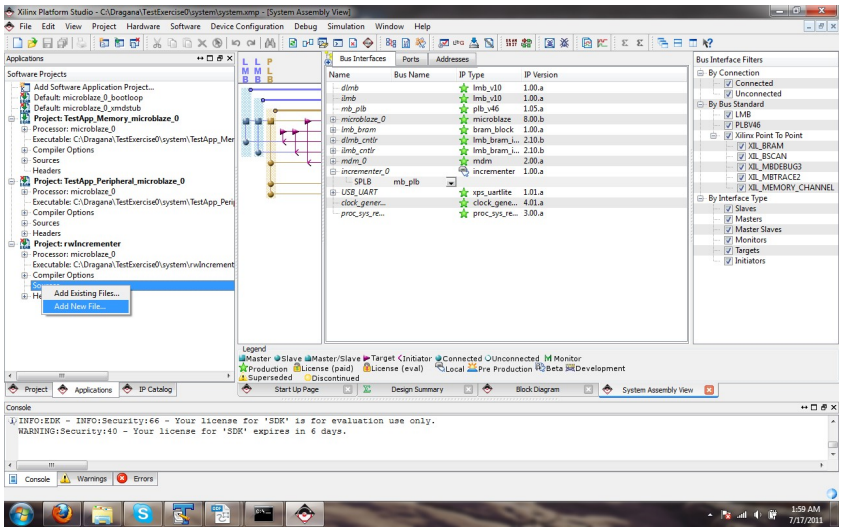

Figure 2.113: Adding a new source file

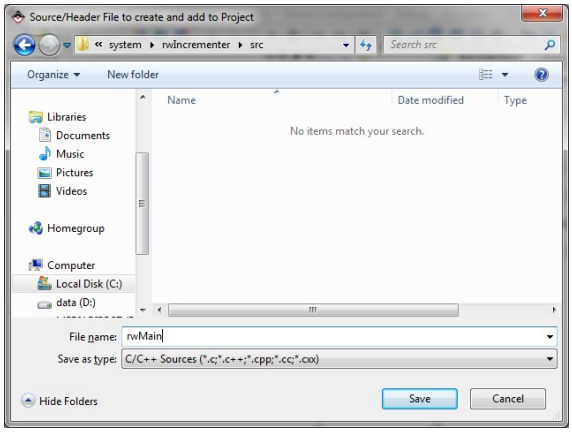

Figure 2.114: Choosing the name for the source file

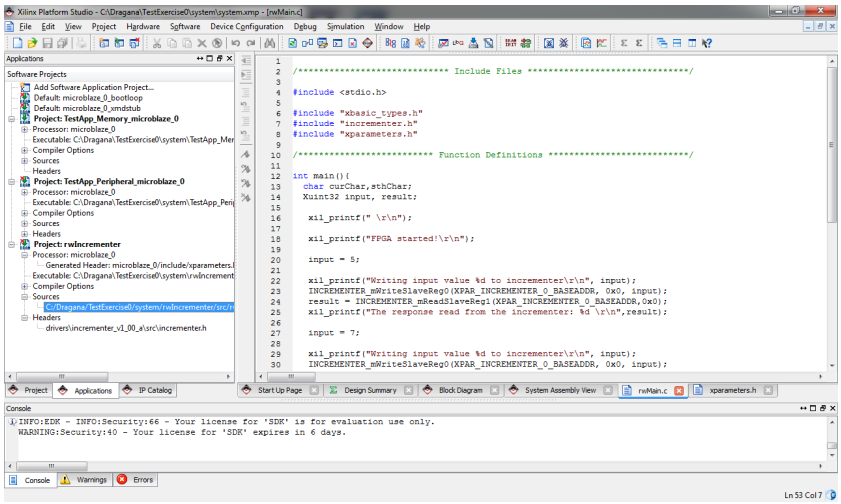

Figure 2.115: The code to be added into the source file

```
/********************** Include Files ******************************/

#include <stdio.h>

#include "xbasic_types.h"
#include "incrementer.h"
#include "xparameters.h"

/******************* Function Definitions **************************/

int main(){
  char curChar,sthChar;
  Xuint32 input, result;

  xil_printf(" \r\n");

  xil_printf("FPGA started!\r\n");

  input = 5;

  xil_printf("Writing input value %d to incrementer\r\n", input);
  INCREMENTER_mWriteSlaveReg0(XPAR_INCREMENTER_0_BASEADDR, 0x0, input);
  result = INCREMENTER_mReadSlaveReg1(XPAR_INCREMENTER_0_BASEADDR,0x0);
  xil_printf("The response read from the incrementer: %d \r\n",result);

  input = 7;

  xil_printf("Writing input value %d to incrementer\r\n", input);
  INCREMENTER_mWriteSlaveReg0(XPAR_INCREMENTER_0_BASEADDR, 0x0, input);
```

```
  result = INCREMENTER_mReadSlaveReg1(XPAR_INCREMENTER_0_BASEADDR,0x0);
  xil_printf("The response read from the incrementer: %d \r\n",result);

  input = 10;

  xil_printf("Writing input value %d to incrementer\r\n", input);
  INCREMENTER_mWriteSlaveReg0(XPAR_INCREMENTER_0_BASEADDR, 0x0, input);
  result = INCREMENTER_mReadSlaveReg1(XPAR_INCREMENTER_0_BASEADDR,0x0);
  xil_printf("The response read from the incrementer: %d \r\n",result);

  xil_printf(" \r\n");

  xil_printf("Now you enter a number between 0 and 9! \r\n");

  do{
          curChar = getchar();
          input = (Xuint32)curChar;
        }while((input < 0) || (input > 9))

  xil_printf("You entered the value %d \r\n", input);
  xil_printf("Passing %d to the incrementer \r\n", input);
  INCREMENTER_mWriteSlaveReg0(XPAR_INCREMENTER_0_BASEADDR, 0x0, input);
  result = INCREMENTER_mReadSlaveReg1(XPAR_INCREMENTER_0_BASEADDR,0x0);
  xil_printf("The response read from the incrementer: %d \r\n",result);

  xil_printf("And this would be all for now! Bye! \r\n");

  xil_printf("\n\n\n");

  return 0;
}
```

It is a simple function which sends an integer number to the implemented design on the FPGA and reads the response from it. If it works as an incrementer (and it should!), for the input number, the response from FPGA will be its successor in the ascending order i.e. the incremented number.

Before you build your project, your compiled files need to be linked with the existing libraries for the hardware platform you are going to use. You will generate a linker script by clicking on the **Generate Linker Script** item on the **Software** menu item as shown in Figure 2.116.

A window opens as in Figure 2.117 in which you are asked to choose the application for which the script is to be generated. Choose the **rwIncrementer** as shown in the figure.

A window opens as in Figure 2.118 in which you are let know that the recommended environment for the software development is Xilinx SDK. However, you will remain working with the software development within XPS environment [1]. So, click **OK** on the notification and stay within the XPS.

A window opens as in Figure 2.119 with the details for the linker as well as the location where the generated files will be saved. Click **OK**.

[1] You are not going to make some extensive software applications for the systems you will develop in this course and learning to use Xilinx SDK would be one more (unnecessary!) task to do in the course

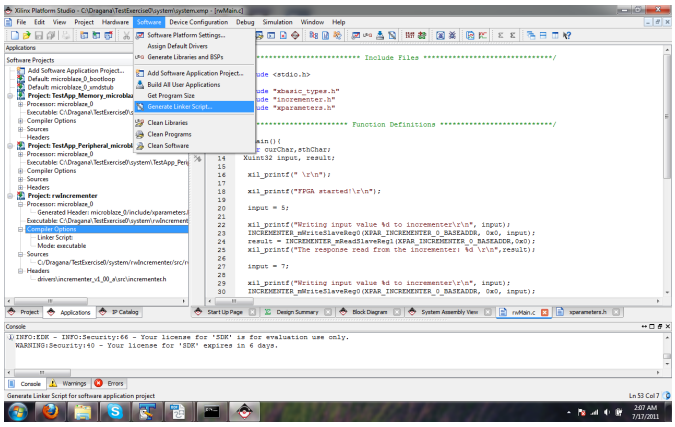

Figure 2.116: Invoking the generation of the linker script

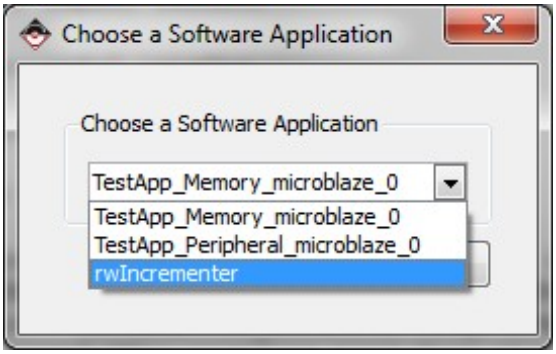

Figure 2.117: Choosing the application for which the linker script will be generated

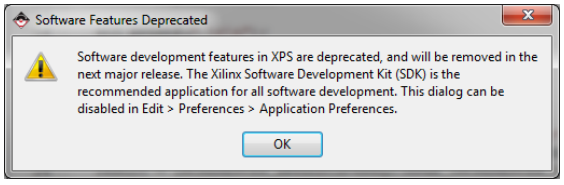

Figure 2.118: Notification on SDK

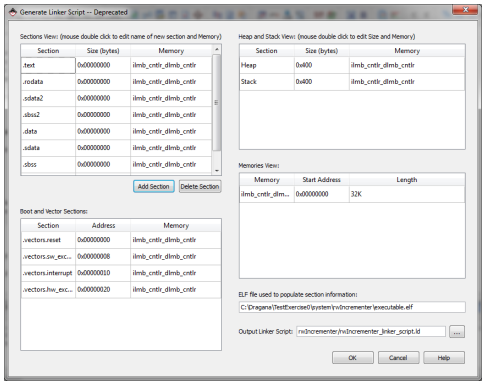

Figure 2.119: Details of the linker script

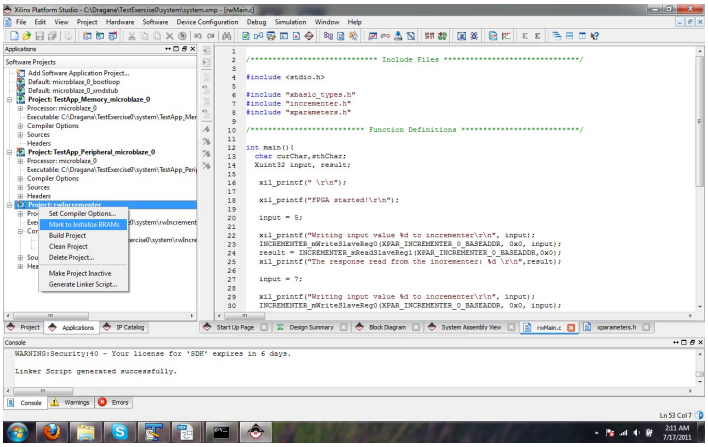

Figure 2.120: Choosing the application which will be downloaded to the BRAM on the chip

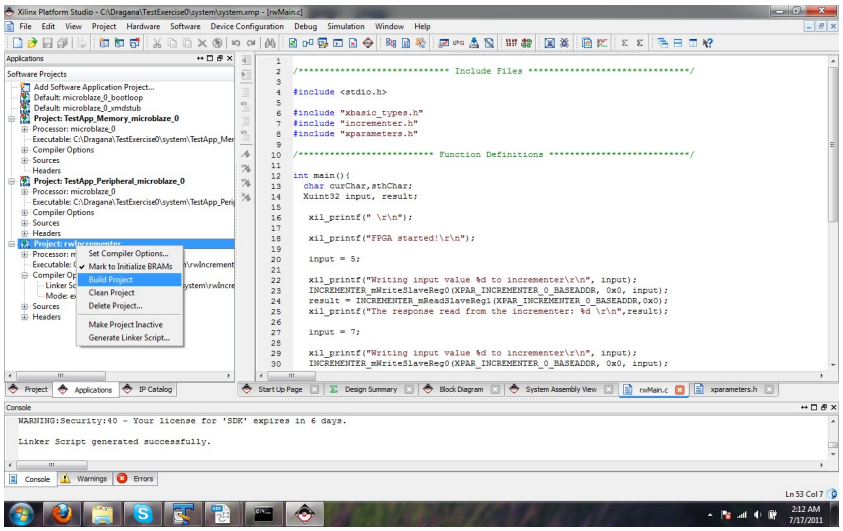

Figure 2.121: Building the project

Currently, it is one of the wizard–generated test applications which would be downloaded to the Spartan–6's BRAM memory with the .bit file. To change it and make your application the one whose .elf file will be merged with the system.bit file, right–click the project name and in the pop–up menu which opens click on the item **Mark to Initialize BRAM** as shown in Figure 2.120.

Click on the **Build Project** item from the same pop–up menu as shown in Figure 2.121. Note that the **Mark to Initialize BRAM** item is now checked.

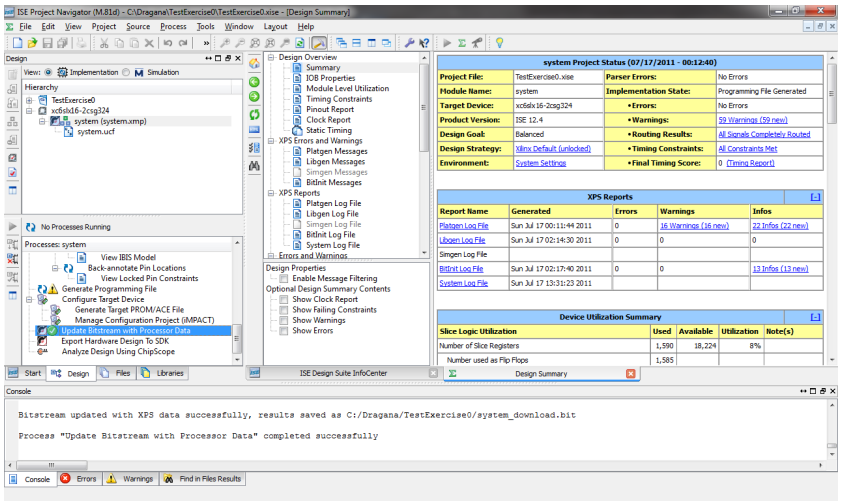

Figure 2.122: Updating the bitstream with the executable of the application

Now exit the EDK environment and go back to the ISE Project Navigator. Click on the **Update Bitstream** item in the **Processes** pane as shown in Figure 2.122. The generated configuration bitstream named **system_download.bit**, with which you will now program the Spartan–6 chip on the Avnet development board, is located in the project folder **<project_ name>\system_download.bit**.

To download the programming file and configure the chip, you will use the Avnet Programming Utility provided by the development board supplier, as further explained.

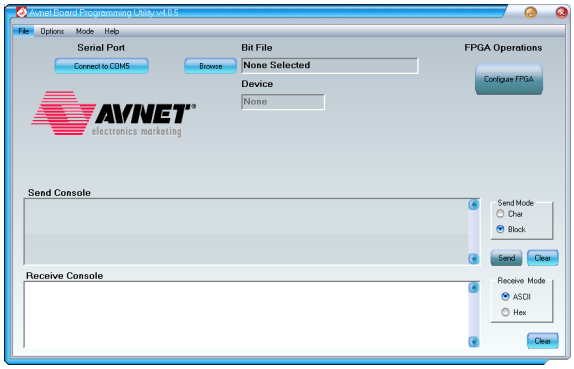

Figure 2.123: Avnet Programming Utility user interface

2.4.3 Avnet Programming Utility

Start the Avnet Programming Utility from the Windows Start menu **Avnet\AvProg**. A window like the one shown in Figure 2.123 opens. The **Send console** and the **Receive console** are used for sending the data to the FPGA and receiving the data from the FPGA respectively. Along the consoles to the right, you can choose one of the options for presenting the data in the consoles. You may choose the mode by clicking to the appropriate radio button. Disabled options and buttons are shown shaded in grey so it is obvious that not much can be done immediately after starting the Avnet Programming Utility. First you need to connect to the board.

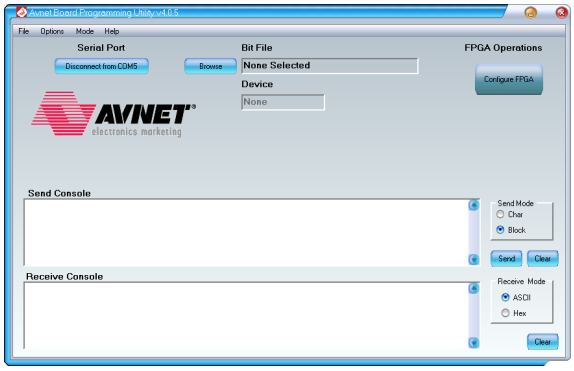

Figure 2.124: Avnet Programming Utility user interface after the connection with the development board has been established

Before connecting to the board, make sure that the development board is physically connected to your PC by the USB cable and that the switch **SW1** on the board is in the position **ON**. The button in the upper left corner of the Avnet Programming Utility shows the option to connect to a COM port which is configured for serial communication with the development board, COM5 in the case shown in Figure 2.123. Click on this button to connect to the board.

Now you are connected to the board and you can access the board through a user interface as shown in Figure 2.124. The default mode is **Configure FPGA** which is exactly what you will need the Avnet Programming Utility for. When connected, the send console is enabled and the button changes into **Disconnect from COMx**. Click on the **Browse** button to locate the .bit file with which you would like to program the FPGA.

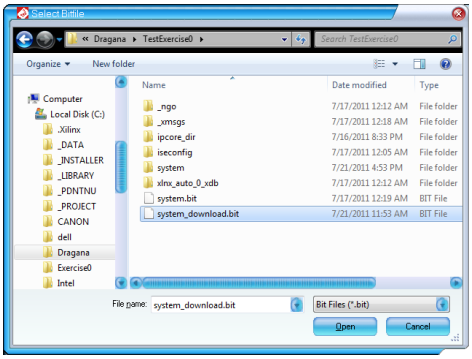

Figure 2.125: Locating the .bit file to be downloaded to the FPGA chip

Locate the **system_download.bit** file at the mentioned location as shown in Figure 2.125

Once the bitfile is chosen, the button **Configure FPGA** in the upper right corner becomes enabled, see Figure 2.126. By clicking this button, the system_download.bit is transfered to the board in order to program the Spartan–6 chip.

Although the device type is selected for you based on the board you are using, you will be asked once again to confirm that this is the right device, as shown in Figure 2.127. Click **Yes**.

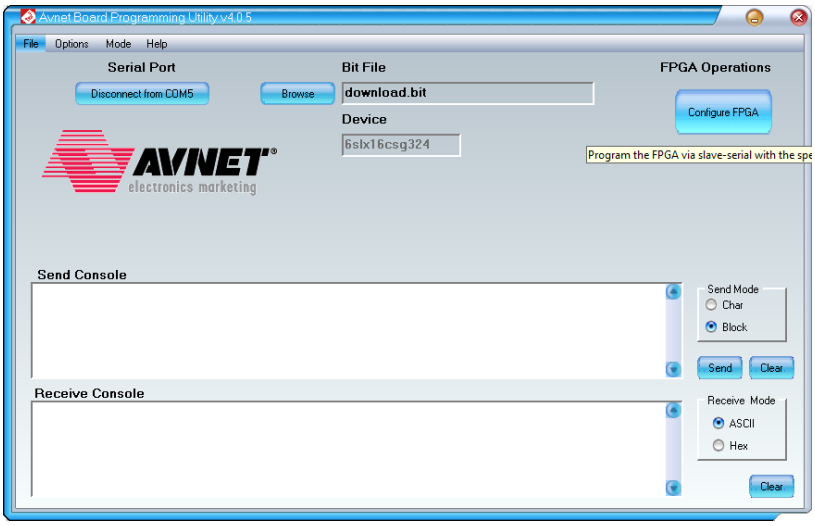

Figure 2.126: Configuring the FPGA: button **Configure FPGA**

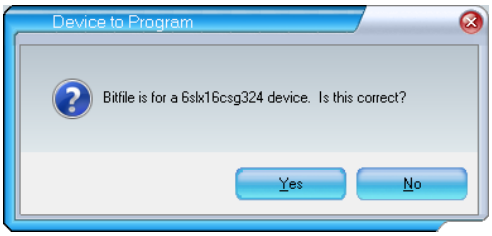

Figure 2.127: Confirmation of the type of the FPGA which is to be configured

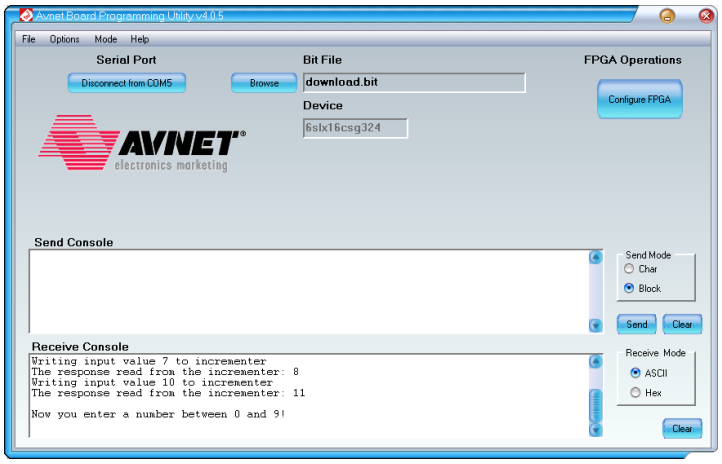

Figure 2.128: The Send and Receive consoles after the FPGA is programmed

After the FPGA is programmed, you receive the notification in the **Receive Console** that 'FPGA has programmed successfully!'. In the same console you may follow the response from your FPGA. When asked to send a number for your incrementer, use the **Send Console** and click the button **Send**.

You may now make modifications to this simple program in XPS. Remember to invoke the option **Update Bitstream** every time after you compile your project in order to merge the newly generated .elf file with your system.bit file into a download.bit file.

Chapter 3

Implementation Framework

3.1 Introduction

In the TDT4255 Computer Design lab exercises, you will be asked to implement a number of processors in three assignments. All of these assignments are based on a common implementation framework. This chapter describes this common framework.

3.2 Implementation Framework

You will test the design in simulation using ModelSim simulator as well as in hardware using a FPGA board. For the implementation in the FPGA, your design will be realised as a peripheral core within a larger embedded system. Figure 3.1 shows which part of the design within an embedded platform you will develop as a peripheral core as explained in Section 2.4.2 and Figure 2.42.

A more detailed block diagram is given in Figure 3.2 which shows the modules provided by the course staff and the processor connected with appropriate signals. The modules delivered with support files are given in parentheses next to the corresponding module.

3.2.1 Communication Module

The inputs to the **com** module are the bus registers that are memory mapped in the Microblaze core so that the **com** module responds to the commands issued by the device driver. The **com** module can write both the instruction and the data memory but only read the data memory. When all necessary data have been loaded into the memories (see the next subsection for how to do it), the device driver issues a command that sets the **processor_enable** signal enabling the processor module which is then given access to the memory modules.

Figure 3.3 shows the state machine of the **com** module. The states change when the device driver changes the value in the memory mapped command register. The three read states stand for the memory read transaction which takes three cycles: the address is provided in cycle 1, the memory is accessed in cycle 2 and the data are stored in an internal register in cycle 3.

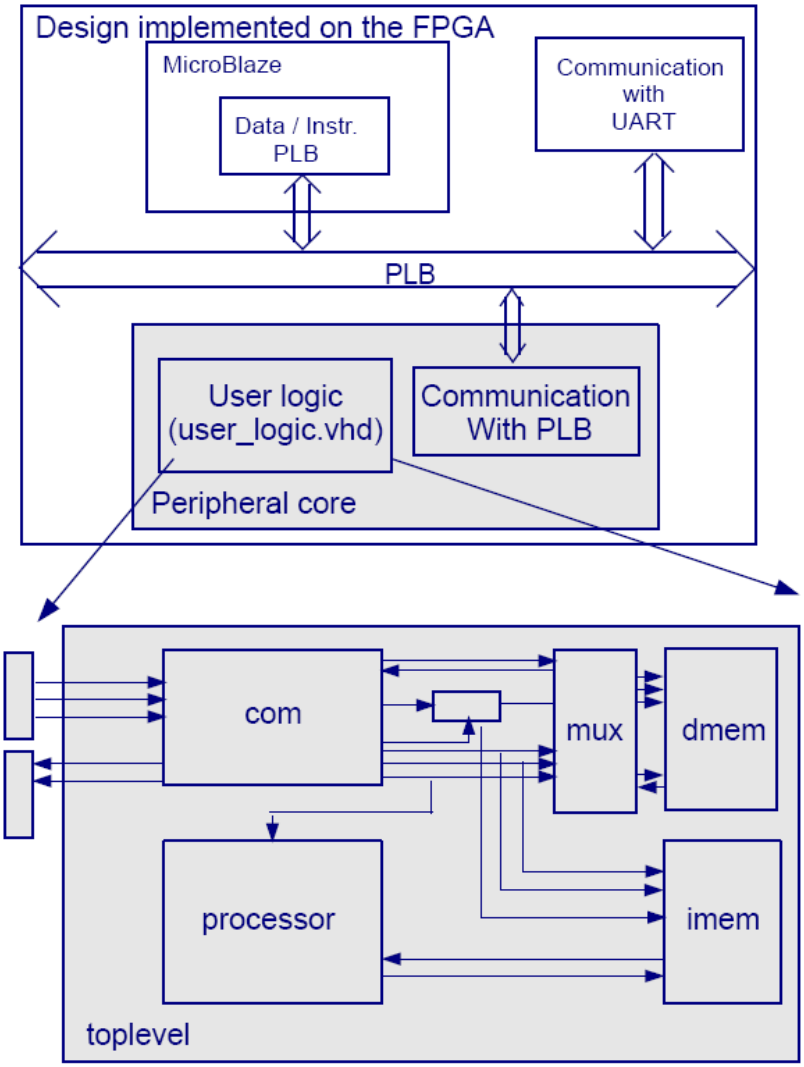

Figure 3.1: The toplevel design which you will implement in all assignments and its relation to the design of peripheral core within an embedded system on FPGA

Table 3.1: Processor Mode Memory Mapped Registers

Register Name	Module Signal	Decimal Offset	Type
Command	command	0	Write Only
Address Input	bus_address_in	4	Write Only
Data Input	bus_data_in	8	Write Only
Status	status	12	Read Only
Data Output	bus_data_out	16	Read Only

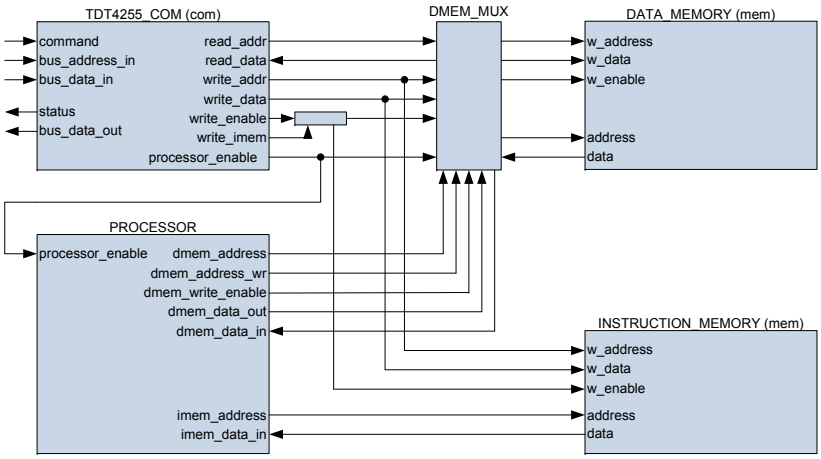

Figure 3.2: Connections between delivered components and the processor component within the toplevel entity. Delivered components are given in parentheses

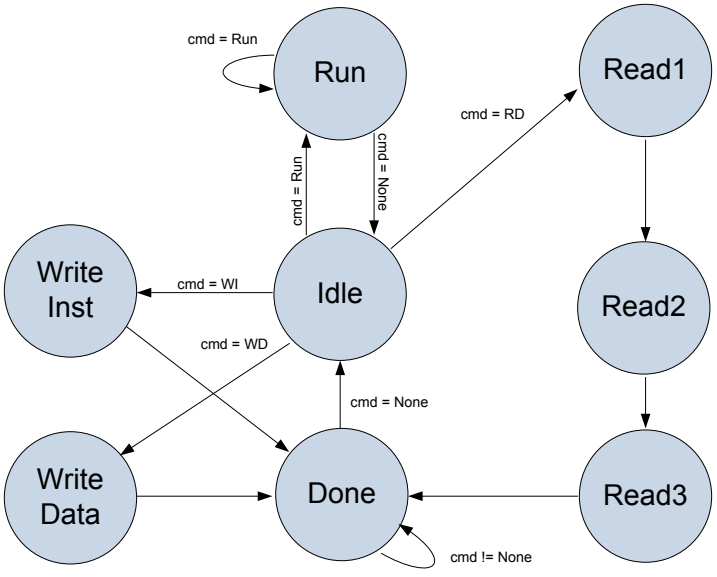

Figure 3.3: Com Module State Machine

Table 3.1 shows the memory mapped registers and which signals they are connected to in the block diagram of Figure 3.2 when the system is in **Processor Mode**. The command encodings are shown in Table 3.2. Since the **com** module state machine changes the state when the command register changes the value, it is important that the data and address registers are written before the command register. Then, the device driver should wait until the **com** module reaches the **Done** state. Listing 3.1 shows an example of how this can be implemented. The **com** module returns to the state **Idle** when the command register is set to **None**. The **com** module also provides some other status information, and the details are available in Table 3.3.

Listing 3.1: Example Busy Wait Loop

```
#define STATUS_DONE 3
void wait(){
  while(TDT4255_EXERCISE2_mReadReg(
            XPAR_TDT4255_EXERCISE2_0_BASEADDR,
            REG_STATUS) != STATUS_DONE){
  }
}
```

3.2.2 Host PC Command Interface

For the communication between the host PC and FPGA, the `host.py` script will be used which provides a simple command interface for several types of transactions. The transactions needed for the assignments are **Write Transactions**, **Read Transactions** and **Command Transactions** and Table 3.4 shows the options which should be in command interface for each of these transactions. After you implement the design in the FPGA, you will write the test program for the processor design into the instruction memory by invoking the `host.py` script with the `-i` option and providing the name of the file which contains the program in a form of a sequence of instructions from the instruction set you have implemented. Analogously, all necessary data will be written to data memory only with the use of `-d` option. To check the results of the processor operation which are written into data memory, you will use `-r` option and provide the name of the file on your host PC to which you would like these data to be written.

3.3 Instruction Set Architecture

In the TDT4255 lab exercises, you will be responsible for implementing a MIPS like Instruction Set Architecture (ISA). MIPS is an acronym for Microprocessor without Interlock Pipeline Stages. MIPS is very popular microprocessor in embedded devices. Instruction words could be set up as follows:

`R-Type:` This group contains all instructions that do not require an immediate value, target offset, memory address displacement, or memory address to specify an operand. This includes arithmetic and logic with all operands in registers, shift instructions, and register direct jump instructions (jalr and jr). All R-type instructions use a 000000 opcode. The operation is specified by the function field.

Table 3.2: Com Module Command Encoding

Command	Mnemonic	Code	Decimal Code Value
No command	None	000	0
Write Instruction Memory	WI	001	1
Read Data Memory	RD	010	2
Write Data Memory	WD	011	3
Run Processor	Run	100	4

Table 3.3: Com Module Status Encoding

Status	Code	Decimal Code Value
Idle	00	0
Busy	01	1
Processor Running	10	2
Done	11	3

Table 3.4: Host script options

Option	Description	Transaction Format
–i <filename>	write the host file <filename> to instruction memory	Memory_ Write
–d <filename>	write the host file <filename> to data memory	Memory_ Write
–r <filename>	read data memory and write to the host file <filename>	Memory_ Read
s	enable/disable processor	Command

Table 3.5: R-Type instruction format

name	opcode	rs	rt	rd	shamt	funct
bits	31–26	25–21	20–16	15–11	10–6	5–0

Table 3.6: I-Type instruction format

name	opcode	rs	rt	immediate
bits	31–26	25–21	20–16	15–0

Table 3.7: J-Type instruction format

name	opcode	target
bits	31–26	25–0

- `opcode`: is the instruction opcode, and function specifies a particular arithmetic operation.
- `rs, rt and rd` : are source and destination registers
- `funct` field used for choosing the instruction's behaviour (ADD, SUB, AND etc.)
- `shamt`: no of bits to be shifted

`I-Type :` This group includes instructions with an immediate operand, branch instructions, and load and store instructions. In the MIPS architecture, all memory accesses are handled by the main processor, so coprocessor load and store instructions are included in this group. All opcodes except 000000, 00001x, and 0100xx are used for I-type instructions

- `rt` is the destination for lw, but a source for beq and sw.
- `imm` is a 16-bit signed constant.

`J-Type:` This group consists of the two direct jump instructions (j and jal). These instructions require a memory address to specify their operand. J-type instructions use opcodes 00001x.

More detailed information about the MIPS architecture is given in Fig.3.4:

3.4 Support Files

Some components are given out within support files which can be used in your design.

- **hardware/toplevel.vhd**: toplevel for your design
- **hardware/com.vhd**: communication module
- **hardware/mem.vhd**: A general memory with parameterisable size which can be used as instruction memory and data memory. The memory is synchronous. That means that the output (data) is available at the next rising

MIPS/SPIM Reference Card

CORE INSTRUCTION SET (INCLUDING PSEUDO INSTRUCTIONS)

NAME	MNE-MON-IC	FOR-MAT	OPERATION (in Verilog)		OPCODE/ FUNCT (Hex)
Add	add	R	R[rd]=R[rs]+R[rt]	(1)	0/20
Add Immediate	addi	I	R[rt]=R[rs]+SignExtImm	(1)(2)	8
Add Imm. Unsigned	addiu	I	R[rt]=R[rs]+SignExtImm	(2)	9
Add Unsigned	addu	R	R[rd]=R[rs]+R[rt]	(2)	0/21
Subtract	sub	R	R[rd]=R[rs]-R[rt]	(1)	0/22
Subtract Unsigned	subu	R	R[rd]=R[rs]-R[rt]		0/23
And	and	R	R[rd]=R[rs]&R[rt]		0/24
And Immediate	andi	I	R[rt]=R[rs]&ZeroExtImm	(3)	c
Nor	nor	R	R[rd]=~(R[rs]|R[rt])		0/27
Or	or	R	R[rd]=R[rs]|R[rt]		0/25
Or Immediate	ori	I	R[rt]=R[rs]|ZeroExtImm	(3)	d
Xor	xor	R	R[rd]=R[rs]^R[rt]		0/26
Xor Immediate	xori	I	R[rt]=R[rs]^ZeroExtImm		e
Shift Left Logical	sll	R	R[rd]=R[rs]≪shamt		0/00
Shift Right Logical	srl	R	R[rd]=R[rs]≫shamt		0/02
Shift Right Arithmetic	sra	R	R[rd]=R[rs]≫>shamt		0/03
Shift Left Logical Var.	sllv	R	R[rd]=R[rs]≪R[rt]		0/04
Shift Right Logical Var.	srlv	R	R[rd]=R[rs]≫R[rt]		0/06
Shift Right Arithmetic Var.	srav	R	R[rd]=R[rs]≫>R[rt]		0/07
Set Less Than	slt	R	R[rd]=(R[rs]<R[rt])?1:0		0/2a
Set Less Than Imm.	slti	I	R[rt]=(R[rs]<SignExtImm)?1:0	(2)	a
Set Less Than Imm. Unsign.	sltiu	I	R[rt]=(R[rs]<SignExtImm)?1:0	(2)(6)	b
Set Less Than Unsigned	sltu	R	R[rd]=(R[rs]<R[rt])?1:0	(6)	0/2b
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5
Branch Less Than	blt	P	if(R[rs]<R[rt]) PC=PC+4+BranchAddr		
Branch Greater Than	bgt	P	if(R[rs]>R[rt]) PC=PC+4+BranchAddr		
Branch Less Than Or Equal	ble	P	if(R[rs]<=R[rt]) PC=PC+4+BranchAddr		
Branch Greater Than Or Equal	bge	P	if(R[rs]>=R[rt]) PC=PC+4+BranchAddr		
Jump	j	J	PC=JumpAddr	(5)	2
Jump And Link	jal	J	R[31]=PC+4; PC=JumpAddr	(5)	2
Jump Register	jr	R	PC=R[rs]		0/08
Jump And Link Register	jalr	R	R[31]=PC+4; PC=R[rs]		0/09
Move	move	P	R[rd]=R[rs]		
Load Byte	lb	I	R[rt]={24'b0, M[R[rs]+ZeroExtImm](7:0)}	(3)	20
Load Byte Unsigned	lbu	I	R[rt]={24'b0, M[R[rs]+SignExtImm](7:0)}	(2)	24
Load Halfword	lh	I	R[rt]={16'b0, M[R[rs]+ZeroExtImm](15:0)}	(3)	25
Load Halfword Unsigned	lhu	I	R[rt]={16'b0, M[R[rs]+SignExtImm](15:0)}	(2)	25
Load Upper Imm.	lui	I	R[rt]={imm,16'b0}		f
Load Word	lw	I	R[rt]=M[R[rs]+SignExtImm]	(2)	23
Load Immediate	li	P	R[rd]=immediate		
Load Address	la	P	R[rd]=immediate		
Store Byte	sb	I	M[R[rs]+SignExtImm] (7:0)=R[rt](7:0)	(2)	28
Store Halfword	sh	I	M[R[rs]+SignExtImm] (15:0)=R[rt](15:0)	(2)	29
Store Word	sw	I	M[R[rs]+SignExtImm]=R[rt]	(2)	2b

REGISTERS

NAME	NMBR	USE	STORE?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes
$f0-$f31	0-31	Floating Point Registers	Yes

(1) May cause overflow exception
(2) SignExtImm = {16{immediate[15]},immediate }
(3) ZeroExtImm = {16{1b'0},immediate }
(4) BranchAddr = {14{immediate[15]},immediate,2'b0 }
(4) JumpAddr = {PC[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2 s comp.)

BASIC INSTRUCTION FORMATS,
FLOATING POINT INSTRUCTION FORMATS

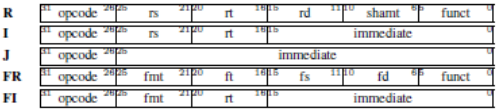

Figure 3.4: MIPS Quick Reference

edge of the clock signal. The output, therefore, acts as a register and there is no reason to make an additional instruction register after it. This register makes it impossible for this processor to be made as a single cycle machine but it needs a separate *fetch*–state to fetch out the instruction from the memory. The memory has a write port which can be directly connected to the **com** module so that the **com** module can write new programs to the memory.

- **hardware/regfile.vhd**: register file
- **hardware/alu.vhd**: a simple Arithmetic Logic Unit (ALU).
- **hardware/processor.vhd**: a file which contains the skeleton for the processor design; you can write your code within designated areas in this file
- **hardware/user_logic.vhd**: this file is provided for the sake of comparison so that you can check the mappings and connections with software accessible registers are correctly modified in the user_logic.vhd file generated by the CIP wizard
- **driver/main.c**: a driver for the peripheral core which corresponds to your processor design
- **driver/host.py**: a script for the host PC command interface

Chapter 4

Assignment 1 – Simple Multi-cycle MIPS Processor

4.1 Introduction

In this assignment, you will design a simple multi-cycle MIPS processor in VHDL and synthesise your design by following the procedure described in Chapter 2. You will also verify the behavior of the implemented MIPS processor using the ModelSim simulator. Once your design in verified, you will integrate the MIPS processor into a MicroBlaze-based embedded system as a peripheral core, implement the embedded system design in FPGA and test the functionalities of the designed processor in an FPGA.

4.2 Requirements

The main requirement for the processor design is a simple multi-cycle MIPS architecture. However, you have to follow the architecture presented in Figure 4.1 for your MIPS processor. Some units ready for the use will be delivered as part of the support files. You may include them into the processor design to build a fully operational processor.

With respect to the instruction set, you have to implement the instructions from each of the the following classes.

- ALU instructions (required to implement minimally ADD, SUB, SLT, AND, OR instructions)
- Conditional branch instruction (BEZ - branch if equal to zero)
- LOAD and STORE instructions
- LDI (Load Immediate - load the register with the given value)
- Jump instruction (J-jump to the specified address)

Once the simple Multi-cycle MIPS processor is designed, you need to verify the functionalities of the designed processor in a simulation environment as well as in a hardware platform.

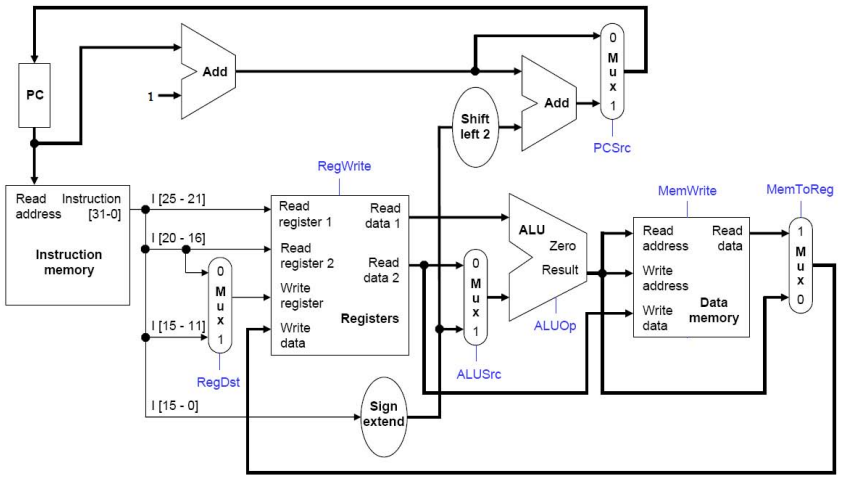

Figure 4.1: Suggested architecture for simple multi-cycle MIPS processor

4.3 Suggested Architecture

The suggested architecture for the simple multi-cycle MIPS processor is depicted in Figure 4.1. Major components of the processor are a program counter, an instruction decoder, a control unit, a register file, a memory module (used to implement both the instruction and data memories), and an ALU. All these modules are implemented individually and then combined to form the MIPS processor. The VHDL implementations of the ALU, the register file and the memory module will be provided as supporting files.

4.3.1 Special Registers

The special registers are:

- Program counter (PC): Contains the address of the instruction which will be fetched from the instruction memory. It must be able to be incremented (increased by one) for every instruction and to be loaded with the new value when a branch instruction is conducted.

- Status register (SR): Contains the status flags from the previous ALU–instruction. This architecture needs only a zero–flag which shows if the previous ALU–instruction gave result 0. It is used together with BNZ to make a conditional branch.

4.3.2 Instruction Word

For this assignment, you should follow the encoding format of MIPS instruction set. The encoding format of the MIPS instructions are described in Section 3.3 in this compendium.

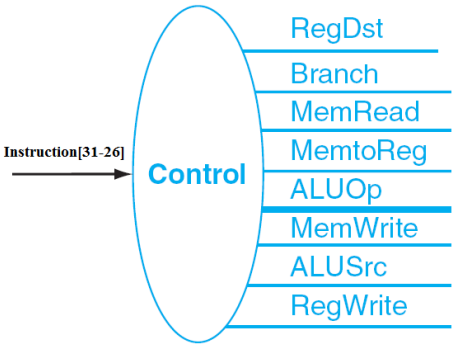

Figure 4.2: Control Unit

4.3.3 Control Unit

Figure 4.2 illustrates the different signals of the control unit of the MIPS processor. As given by the Figure, the control unit should have possibility to choose inputs to both multiplexors, control *write enable* for PC and SR and control *write enable* for the register file.

Because memory access takes the clock cycle, we need to implement the Control unit as a state machine with the following states:

- Fetch: Fetch instruction from the instruction memory
- Execute: Decode and run the instruction and write the result back to the register file. Decoding in the execute state can consist of a CASE–statement which determines behaviour based on the opcode. The behaviour will contain setting the control signals for the rest of the processor.
- Stall: Since the memory access causes additional latency(one extra cycle), for some instructions e.g. LOAD, STORE it will require more than one cycle to complete the execution of such instruction. Which entails that you may need to define a new state for the processor.

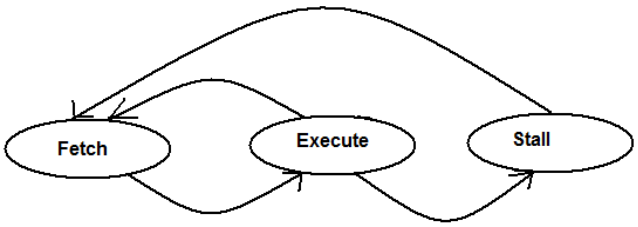

Figure 4.3: Example for the control unit state machine

Chapter 5

Assignment 2 – A Simple Pipelined Processor

5.1 Introduction

In the second assignment, you will extend the processor from previous assignment by changing the datapath to a pipeline. This means that you will need to add pipeline registers and make a new control module which accommodates the pipeline processing. **Note that all hazards are handled through software**. Therefore, in this assignment, you don't need to implement the hazard detection and controlling module. Hazard detection and controlling module using hardware is left for Assignment 3.

5.2 Requirements

The major requirement of this assignment is a simple 5-stage pipelined processor. In general, the processor has the same functional requirements as in the previous assignment. You will also use the same test set up. To help you on the way, we have made a suggestion from which you can work on. It is wise to make a processor which relies on the design from the previous assignment so that you can reuse the test benches and test programs.

5.3 Suggestion for the Architecture

We suggest you to follow the architecture presented by Patterson and Hennessy in [1] for the simple pipelined processor. The architecture given by Patterson and Hennessy is presented in Fig. 5.1. This is a five stage pipeline processor which is a natural extension of Assignment 1. Here we have made a data storage (DMEM). In addition, we have added four pipeline registers (IF/ID, ID/EX, EX/MEM and MEM/WB). Data storage is addressed with the immediate field in the instruction word and it has register A connected to the data input.

The Control unit (CONTROL in the figure) is placed in the decode stage. It is a combinatorial circuit and not a state machine as in the previous assignment. The control entity will now, based on the opcode and status register, set up the

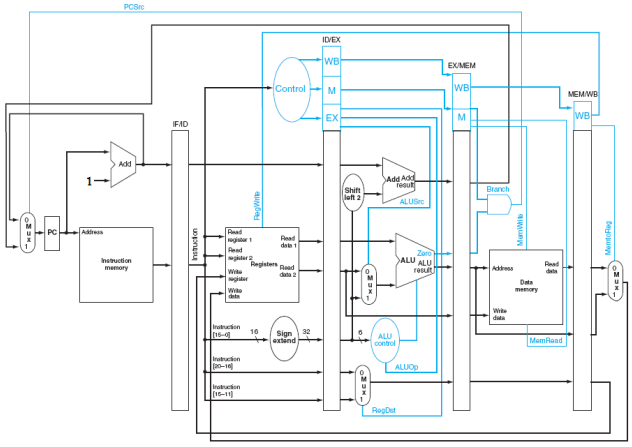

Figure 5.1: Suggested architecture

control signals for all pipe stages. Control signals for later pipeline stages are sent to pipeline registers. In Fig.5.1, these signals are blue.

Branch instructions do not need to be sent through the whole pipeline, but in this architecture they can be taken already to the decode stage. If the branch should be taken, the control entity can set the program counter's multiplexor so that the immediate field in the instruction word is loaded into the PC.

Chapter 6

Assignment 3 – Optimized Pipelined Processor

6.1 Introduction

In the last assignment, you will extend the previously implemented pipelined processor to optimize its performance by implementing different hazard detection and correction techniques. Some of these techniques include, but are not limited to, data forwarding and pipeline interlocks that stall the pipeline when necessary. In addition, you can implement different optimization techniques to improve the performance of your pipelined processor.

6.2 Requirements

In general, the processor has the same functional requirements as in the previous assignment. Additionally, you need to implement different hazards detection and correction techniques. It is wise to make a processor which relies on the design from previous assignment so that you can reuse the test benches and test programs.

Appendix A

The List of Versions

Here is the list of the compendium revisions:

- Version 1, 2011-08-31: New version for 2011
- Version 2, 2012-04-30: New version for 2012

Bibliography

[1] David A. Patterson and John L. Hennessy. *Computer Organization and Design*. Elsevier, 2005.

[2] Xilinx. *XST User Guide*, 2008.

[3] Xilinx. *EDK Concepts, Tools and Techniques*, 2010.

[4] Xilinx. *ISE In–Depth Tutorial*, 2010.

[5] Xilinx. *Spartan-6 Family Overview*, 2011.

[6] Xilinx. *Spartan-6 FPGA Block RAM Resources, User Guide*, 2011.

[7] Xilinx. *Spartan-6 FPGA Configurable Logic Block, User Guide*, 2011.