Guide to Synthesis and Implementation Tools for VHDL Modeling and Design[1]

Martin Rosner

ECE Department
Worcester Polytechnic Institute
Worcester, MA 01609-2280
mrosner@ece.wpi.edu

[1]Internal document CRIS-WPI0001

Table of Contents

List of Figures

Chapter 1

Overview

1.1 About This Manual

During the course of my MS thesis I had the less than pleasant opportunity to learn the tools used for VHDL synthesis, simulation and implementation. Together with a colleague of mine, Jens-Peter Kaps, we managed to set up libraries and run the appropriate tools in order to realize our designs. In this case, the word 'realize' encompasses many stages of the development for which different tools had to be installed and utilized. The generous help of our sys-admin guys, Brady Schulman and Murtaza Amiji was crucial during this setup and installation stage. In this document, I will try to describe the process that we used to realize a conceivable digital design. This document is not intended to be 100% thorough and should be used as an initial guide to the synthesis, simulation and implementation tools. Most of the information presented here can be found in the online documentation provided by the vendors. My job here is just to summarize the most important and the hard to find information regarding the setup and utilization of these tools. For a more formal and more complete source of information, the reader is directed to the vast number of manuals that exist and are available online or in the crypto lab. Whenever appropriate, I will try to point out the particular documentation manual and describes a certain subject in greater detail.

In this chapter, I will give a general description of the process that we used to realize our designs. I will also briefly describe our main method of design entry and some of the more important considerations associated with VHDL entry and synthesis. Each consecutive chapter after this one will describe one step of the process and the tools associated with this step. Thus if you just want to learn about synthesis you should refer to Chapter 3 that describes the Synopsys tools that we used to perform synthesis.

1.2 The Process

The general design cycle that was adopted during my work will be described in detail in this manual. Assuming that the designer has conceived a digital design the next series of steps that will realize such design are described as follows:

1. VHDL design entry.
2. Logic verification of the design (RTL simulation).
3. Synthesis and logic optimization.
4. Device specific realization (place and route).
5. Back-annotated verification of the design (PPR simulation).
6. Optimization
7. Implementation

Because of the vast array of reconfigurable devices available today, VHDL implementations tend to become vendor specific. This is due to the fact that many vendors provide soft macros for predefined components. These macros take advantage of specific features in a given device making it nearly senseless to build equivalent functions with VHDL. By choosing a specific family of devices, it was now possible to identify useful macros for the conceived design. Having the knowledge of these macros and the initial design, a digital system can be developed in VHDL with macro instantiations. This process requires the use of special libraries and will be described for the simulation and synthesis chapters of this manual.

Verification of the design is first performed on the logic level basis. This step assures the correct functionality of the design without the combinatorial and net delays. Once the design is verified logically, synthesis and optimization is performed. Timing constraints are set for each component and different iterations are done until constraints are met. The next step is to actually map, place and route the design into reconfigurable device. The choice of a specific device within the device family depends on the area utilization report obtained through synthesis. Finally, the output of the place and route step is used to perform back-annotated simulation. This step verifies the correct operation with net and combinatorial delays that results from the place and route process.

1.2.1 Tools

The entire design, with the exception of vendor specific soft macros, can be entered in VHDL format. Once the design is developed in VHDL, boolean logic and major timing errors are verified by simulating the gate level description with Synopsys VHDL analyzer (`vhdlan`) version 1997.08. The next step involves synthesis of the VHDL code with Synopsys (`fpga_analyzer`) version 1997.08. The output of this step is an optimized netlist describing the gate level design in XILINX format. The most time consuming step is the compilation of the synthesized design with the place and route tools available from XILINX. The slow compile times associated with this step may result in a limited amount of data that could be gathered during this process. This step is accomplished with the XILINX Design Manager tools version M1.3.7.

The next step of the design flow is to verify the design once again but this time with the physical net, CLB, and pad delays introduced when the design was placed into a specific device. This final stage of the design is accomplished with the same test benches and simulation models that are used during the logic verification stage. Synopsys (`vhdlan`) is used once again to verify back-annotated designs.

Xilinx Synopsys Interface

Figure 1.1 presents a flow chart diagram of the design flow with Xilinx-Synopsys-Interface (XSI) tools. The XSI tools provide for a transition between results obtained from Synopsys synthesis and the Xilinx place and route tools. The XSI module includes all libraries necessary for Synopsys `fpga_analyzer` to interpret gates into logical blocks so that synthesis can be performed at this level. The design_ware libraries

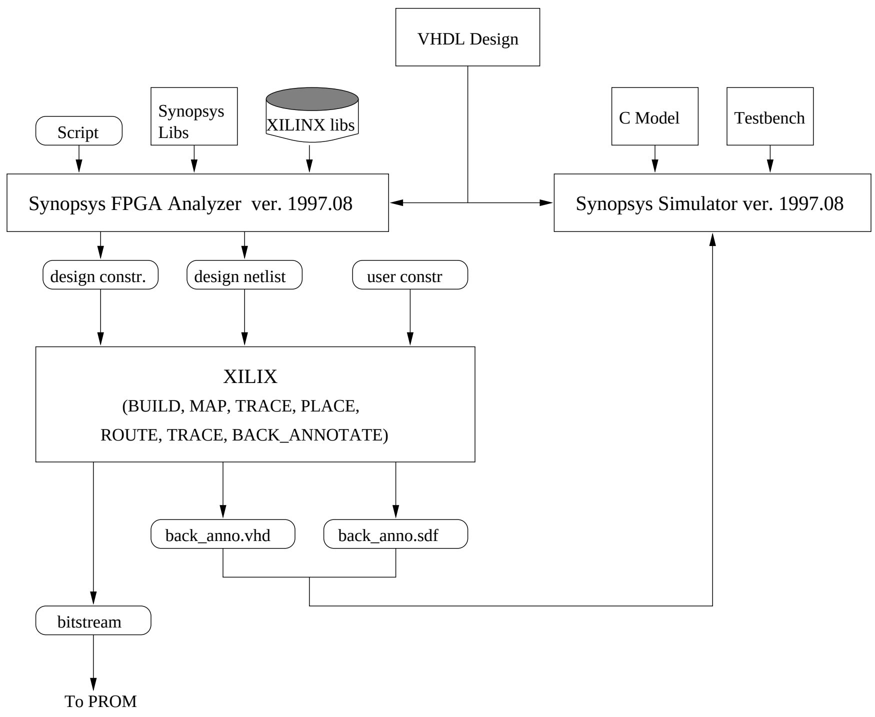

Figure 1.1: Design flow

provided by Xilinx are automatically instantiated when possible. For example, if the

VHDL code contains addition "+" operations (such as the one used to increment counters), the Synopsys tool will utilize Xilinx design_ware libraries to instantiate soft macros for such adder units. Synthesis results include report files on area and timing utilization, design netlist and constraints that are used in the place and route process, and Synopsys design files that describe the entire system.

Simulation and Verification (RTL)

As previously stated, verification of the design is done at two points. First, it is applied to the initial VHDL design. This process is called the Register Transfer Level (RTL) simulation. This verifies only the logic without delays. The input to this verification process is a test bench written in VHDL, a model of the design written in C, and the actual VHDL design. The test bench is used together with the VHDL design to simulate the design. Then the results from the simulation are compared against results obtained from the C model. Logical verification is complete when all the test vectors are verified. This process is described in detail in Chapter 2.

Synthesis

Synopsys synthesis tools have only been available to us in the past year. The documentation that accompanied these tools is quite extensive and very helpful. This and other literature helped in developing script files that could be launched from within the `fpga_analyzer`. These scripts would elaborate, compile, optimize the design, and prepare report summaries. One advantage of running this tool on an HP station is that multiple jobs could be run concurrently resulting in faster turnaround and more time to try different optimization options. The synthesis process is described in detail in Chapter 3.

Place and Route

The place and route tools from Xilinx can be used on the HP workstation as well. Although by the time I leave WPI, I think that license will run out on the HPs. Besides, the compilation proved to be very slow is complex routing is required. Consequently, this step can be moved to a Windows environment so fast Pentium-based PCs could be used. This is not to say that this tool runs better in Windows. It is just that the availability of better hardware forced the migration to the Windows environment.

The input to the place and route tools is a design netlist and constraints files generated by Synopsys, as well as possible user constraints file. The user constraints have higher priority over the Synopsys constraints and may include additional constraints relaxing the clock period or implementing pin assignment. As stated before, the output of this process is a bit-stream file that can be used to directly program the device and the back-annotated design that can be simulated for timing verification (please refer to Figure 1.1). This process is described in detail in Chapter 4.

Simulation and Verification (PPR)

The Post-Place-and-Route (PPR) simulation and verification uses the same C model and test bench (with few modifications). The VHDL input model to this stage is different. Here the VHDL model is obtained from the XILINX place and route tools. This VHDL model looks nothing like the original one and it includes a separate file defining all net, CLB, and port delays associated with the placed design. Once again, verification process involves testing all vectors against the C model results using the Synopsys (`vhdlan`) tools. This process is described in detail in Chapter 5.

Hardware Realization

This final step of the implementation process requires actual hardware components that are used to program and run the device. This step was not a goal for either of our thesis. Consequently I will leave the chapter describing this step blank. Perhaps the MQP group that worked on the prototype board can fill this chapter in.

1.3 Design Entry Considerations

The popularity of reconfigurable hardware is growing for many reasons. Most importantly such design approach yields itself very nicely to fast turnaround during the prototyping stage and allows scalability as well as modularity to the fullest extent. VHDL has become the standard hardware description language almost entirely abstracting the hardware designer from the actual hardware. Consequently, the necessity to understand how this design entry method effects synthesis and simulation is crucial to efficient design. During several projects, both Jens and I encountered some issues associated with the interpretation of specific VHDL primitives into a not so efficient digital components. By pointing some of these out in this section I hope only to show you that VHDL design entry must be carefully executed to ensure a solid digital realization.

Perhaps one of the most important rules in VHDL design is to define all conditions. Both Jens and I have noticed that when we define all possibly conditions on a bus that is used in the `CASE-WHEN` structure, the logic created is much smaller and faster. This occurs because additional flip-flops are necessary if not all conditions are defined. Since VHDL abstracts the designer from the design, it is sometimes very hard to consider all cases of conditional logic resulting in a less than optimal design.

Another important issue when dealing with VHDL and re-programmable device is the fact that asynchronous design does not yield itself very nicely to FPGAs or any other re-programmable devices. Although synchronous designs are more popular and easier to implement, asynchronous designs exhibit properties that are desirable in many digital systems. One major attribute of asynchronous designs is that handshaking between processing elements is more defined absolving the need for a centralized clock signal. Conversely, synchronous designs rely on a clock signal whose period must be greater than the longest combinatorial delay. Some other more prominent advantages of asynchronous designs are presented in [5]. However, asynchronous design have many disadvantages when considered in the context of FPGA design. As much as we would like to attempt a design of an (EC) digital system using asynchronous methodology, [3] suggests that such an approach necessitates elimination of all hazards, synchronization of events and very precise timing requirements of all functional blocks in the design. Timing glitches due to poor combinatorial design can be avoided with careful consideration. However, timing glitches produced by the place and route tools are, for the most part, very hard to control. It is these glitches that render asynchronous designs on FPGAs infeasible at this point. Few attempts have been made to develop reconfigurable devices capable of asynchronous operation [3, 4]. Thus, until such devices become readily accessible, FPGA implementations are limited to synchronous methodology.

The means for implementing state machines in the XC4000 family devices is another consideration that needs some elaboration at this point. In particular the state machine encoding style has to be chosen to fully utilize the specific FPGA resources. In a conventional ASIC design, binary or gray code state machine encoding is preferred for efficient and minimal design [6]. Conversely, "one-hot encoded" state machines implemented in ASIC device exhibit relatively large area with minimal gain in performance. This is not the case with FPGA devices. As described in [2], one-hot encoding becomes feasible when mapped to FPGAs due to the large number of flip-flops available in the device. For many designs, the CLBs are used as LUTs leaving the flip-flops untouched. Consequently, these resources can be utilized when one hot encoded state machine is implemented [2]. The advantage of a one-hot encoding lies in the representation of each state with an individual bit (flip-flop) resulting in decreased logic complexity associated with each state [1]. However, any FPGA device has a limited number of routing and area resources thus implementing a one-hot encoded state machine is advantageous only in cases where there are enough resources for the state machine and the rest of the design. Once the state machine becomes too complex, [1] suggests that an external RAM may be required to store the control program leaving only the sequencing mechanism inside the FPGA.

Many FPGA manufacturers provide an array of arithmetic and storage macros that take advantage of specific features in a particular device. Xilinx provides a design-ware library filled with such macros in addition to LogiBlox components that can be tailored to the designer's specific need. Arithmetic functions such as adders use the fast carry chains available on the device. Also RAM/ROM elements mapped with available macros fully utilize the CLB structure to achieve the most performance out of a single CLB. In [2], many designs are analyzed to show the effectiveness of the available macro functions. Our implementation takes advantage of the LogiBlox functions for storage (RAM and registers) elements. Also, the design-ware library is invoked by Synopsys tools to implement arithmetic functions necessary for counters present in the control module.

1.4 Other Sources of Documentation

As mentioned previously, this manual provides brief explanations of the entire process. For detailed help on individual tools, the reader should refer to the help menus associated with these tools. Furthermore, both Xilinx and Synopsys maintain a very helpful WWW database of commonly asked questions, design considerations, VHDL issues, device descriptions, and much more. Please take advantage of these resources as they are updated regularly and contain information that tremendously help both of us during our projects. In addition, there are few manuals in the Crypto lab that may be of help. These manuals are available on WWW, but we printed them because we felt that they were quite important. These are *Xilinx Synopsys Interface FPGA User Guide* and *HDL Synthesis for FPGAs*. Finally, the Synopsys On-line Documentation (SOLD) is probably the most robust reference along with the Xilinx DynaText Help.

1.5 User Directory Setup

Before you begin synthesis or simulation, make sure that the following path definitions and environment variables are defined in your login script:

```
setenv XILINX /local/xilinx
setenv XILINX_PATHLIMIT 100000
setenv LM_LICENSE_FILE /local/xilinx/data/license.dat
setenv XILINX_CD /cdrom
setenv CDG_VHDLGEN /local/synopsys
setenv XKEYSYMBD /local/xilinx/bin/hp/XKeysymDB
setenv EBTRC /local/xilinx/bin/hp/ebtrc
setenv SYNOPSYS /local/synopsys
setenv SYNOPSYSPATH $SYNOPSYS/hpux10/syn/bin/
source $SYNOPSYS/admin/install/sim/bin/environ.csh
set path=($path /local/synopsys/hpux10/syn/bin)
set path=($path /local/xilinx/bin/hp)
set path=($path /local/synopsys/hpux10/syn/bin)
set path=($path /local/synopsys/hpux10/sim/bin)
set path=($path /local/xilinx/bin/hp)
set path=($path /local/synopsys/hpux10/license/bin)
set path=($path /local/synopsys/hpux10/)
setenv SHLIB_PATH /local/orca/bin/hp:/local/synopsys/hpux10/sim/lib:.:/local/synopsys/hpux10/sim/lib:$XILINX/bin/hp:$XILINX_CD/bin/$
setenv LD_LIBRARY_PATH /local/synopsys/libraries:/local/xilinx/synopsys/libraries/
```

Chapter 2

RTL Verification

Register Transfer Level (RTL) simulation and verification is one of the initial steps that was done. This step ensures that the design is logically correct and without major timing errors. It is advantageous to perform this step, especially in the early stages of the design, because long synthesis and place-and-route times can be avoided when an error is discovered at this stage. This step also eliminates all the syntax errors from your VHDL code.

We used the Synopsys simulation tools to perform RTL verification. More specifically, `vhdlan`, or VHDL_analyzer was used. In addition, if the design contains LogiBloX components, (which is usually the case), the corresponding VHDL primitives generated by Xilinx tools along with Xilinx libraries have to be visible to the Synopsys tools. In the following I will describe the process of compiling/simulating a VHDL design that includes LogiBloX using VHDL_analyzer. An example of a test-bench VHDL file will be explained and simulated against the design. Finally, few debugging tips that we picked up along the way will be given.

2.1 Setup Conditions

The setup conditions are defined by a .synopsys_vss.setup file that defines the name of your working directory, time resolution, and many other information. The information included here may also define device/vendor specific libraries that are used during simulation purposes to model a macro component such as a LogiBloX component. A sample of the setup file that was used is shown below:

```
TIMEBASE         = NS
TIME_RES_FACTOR = 1
NO_HAZARD_MESG = TRUE

WORK            > DEFAULT
DEFAULT         : ./WORK

CFLAGS  = -Dhp700Architecture
LDFLAGS = /bin/ld -b -o %s

XC4000e : /tmp/sim2/lib/xc4000e/ftgs
```

It is necessary to ensure that the libraries that are used during simulation (such as the FTGS library shown in the example) is compiled with the same version tool as the current design. For this reason, it may be required to recompile the library source code that can be found in the $XILINX path. The above setup file shows only the most essential information. For a detailed description of definitions that can be used in this file please refer to the Synopsys OnLine Documentation (SOLD).

2.2 Generating LogiBloX Primitives

LogiBloX components were the most widely used macros in our designs. With the available tools from Xilinx, it is possible to generate netlist files as well as simulation primitives for storage components, ALUs, shift registers and many other function blocks. By using LogiBloX components for a specific function, you can ensure most optimal implementation of that function for a specific device. Since LogiBlox modules generate your netlist, synthesis of these components is not necessary.

In order to verify a design that contains LogiBloX, it is necessary to use the logiblox_design.vhd file that is generated by the Xilinx tools. Figure 2.1 shows the setup taplate used to create LogiBloX components. It is essential that the user creates

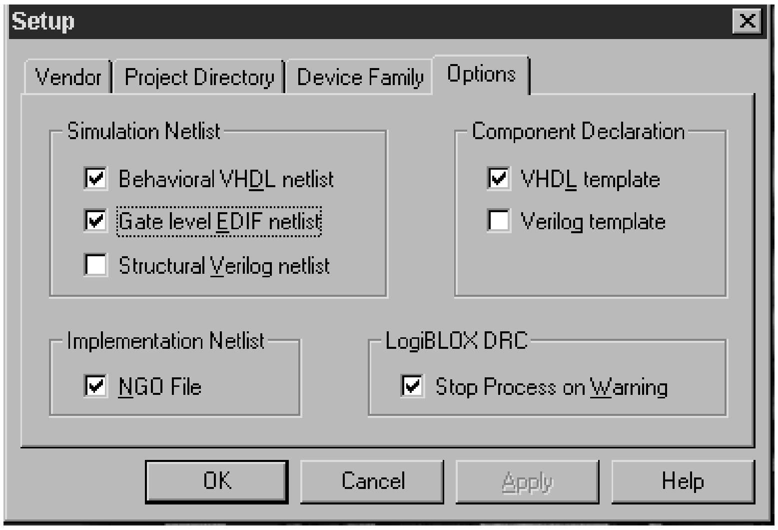

Figure 2.1: LogiBloX setup template

the simulation netlist in the for of a VHDL file and an implementation netlist as the *.ngo file. The simulation netlist is then used with the Synopsys simulator as described in Section 2.4. Similarly, the implementation netlist (*.ngo) is used during the place-and-route process to phisicaly realize desired functionality.

2.3 Testbench Requirements

Synopsys VHDL simulation tools (`vhdlan` and `vhdldbx`) can simulate VHDL testbench files. This was found to be the easiest way in which to apply stimulus during our projects. The most primitive test-bench file may contain a clock generator, design instantiation, and manual stimulus assignments. The test-bench shown below instantiates the final design and applies the stimulus manualy:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_textio.all;
use work.array_types.all;

--
-- VHDL Architecture Test-bench
--
-- Created:
--          by - mrosner.mrosner (pike.WPI.EDU)
--          at - 16:45:58 09/15/97
--
entity SYSTEM_TESTBENCH is
--ENTITY OF A TEST-BENCH IS EMPTY
end SYSTEM_TESTBENCH;

ARCHITECTURE TEST OF SYSTEM_TESTBENCH IS

-- Architecture declarations
        CONSTANT clk_prd : time := 50 ns;
        constant width            : positive := 8; -- bits in signle register
        constant slices           : positive := 21; -- number of registers in array
        constant depth            : positive := 4;
        constant trinomial_coeff  : positive := 2; -- (k) from p. 158 in Menezes
        constant block_size       : positive := 7;

        SIGNAL clk             : std_logic ;
        SIGNAL reset           : std_logic ;
SIGNAL START_JOB : std_logic;
        SIGNAL input_coord_pin  : input_slice_array;
SIGNAL mult_vector : std_logic_vector((slices*width)-1 DOWNTO 0);

        SIGNAL ram_out          : slice_array;
SIGNAL ready  : std_logic ;

        SIGNAL iclk : std_logic;

        PROCEDURE wait_clock(CONSTANT clk_ticks:integer) IS
        VARIABLE i : integer := 0;
        BEGIN
                FOR i IN 1 TO clk_ticks*2 LOOP
                        WAIT UNTIL iclk'EVENT;
                END LOOP;
        END wait_clock;


        component system
                PORT(
                        clk             : IN     std_logic ;
                        reset           : IN     std_logic ;
                START_JOB : IN std_logic;
                        input_coord_pin : IN    input_slice_array;
                mult_vector : IN std_logic_vector((slices*width)-1 DOWNTO 0);

                        ram_out         : OUT slice_array;
                ready  : OUT std_logic
```

```
            );
      end component;

BEGIN
   UUT : system
            Port Map (
                        clk => clk,
                        reset =>reset,
                  START_JOB =>START_JOB,
                        input_coord_pin => input_coord_pin,
                  mult_vector =>mult_vector,

                        ram_out =>ram_out,
                  ready =>ready
                   );

   flow_process: PROCESS

   -- Process declarations
   VARIABLE j          : integer := 0;
   VARIABLE d_var      : unsigned(9 DOWNTO 0) := "0000000000";
   VARIABLE b_var      : unsigned(3 DOWNTO 0) := "0000";

   BEGIN
      input_coord_pin(0) <= "00110011";
      input_coord_pin(1) <= "00110011";
      input_coord_pin(2) <= "00110011";
      input_coord_pin(3) <= "00110011";
      input_coord_pin(4) <= "00110011";
      input_coord_pin(5) <= "00110011";
      input_coord_pin(6) <= "00110011";


      mult_vector <= "1011011010110110101101101011011010110110101101101
                      0110110101101101011011010110110101101101011011010
                      1101101011011010110110101101101011011010110110101
                      101101011011010110110";
      reset <= '1';
      wait_clock(1);
      reset <= '0';
      START_JOB <= '1';
      wait_clock(1);
      START_JOB <= '0';
      wait_clock(6);
      input_coord_pin(0) <= "10011001";
      input_coord_pin(1) <= "10011001";
      input_coord_pin(2) <= "10011001";
      input_coord_pin(3) <= "10011001";
      input_coord_pin(4) <= "10011001";
      input_coord_pin(5) <= "10011001";
      input_coord_pin(6) <= "10011001";

      wait_clock(6);
      input_coord_pin(0) <= "01110111";
      input_coord_pin(1) <= "01110111";
      input_coord_pin(2) <= "01110111";
      input_coord_pin(3) <= "01110111";
      input_coord_pin(4) <= "01110111";
      input_coord_pin(5) <= "01110111";
      input_coord_pin(6) <= "01110111";
      wait_clock(3);

      wait;
   END PROCESS flow_process;
```

```
        -- Architecture concurrent statements
        clock_gen : PROCESS
        BEGIN
                iclk <= '0';
                WAIT FOR clk_prd/2;
                iclk <= '1';
                WAIT FOR clk_prd/2;
        END PROCESS clock_gen;
   clk <= iclk;

END TEST;
```

The other option is to write a test-bench the read stimulus from a file and assigns it to corresponding signals. Furthermore, the test-bench can compare the output of the stimulus to results gathered through different means. In our case, the designs were modeled in C and the results from the C simulation were compared against the test-bench results.

2.4 Compiling Simulation

Simulation can be done in interpreted or compiled mode. Compiled mode requires the C Developers bundle for HP that was not available to us. In compiled mode simulation, debugging becomes faster and easier because the VHDL design is converted to a C model. Consequently, we used the `-i` option to compile our VHDL design in interpretive mode. Using this mode you need to ensure that you also compile `logiblox.vhd`, `mvlarith.vhd`, and `mvlutil.vhd`. files when using LogiBloX component in the design These are library files that have to be compiled in the iterpreted mode as well. Before you can compile these library files, you need to change the library declarations in your LogiBloX simulation netlist, and the library files. For example, the simulation netlist originaly tries to call the `logiblox.vhd`, `mvlarith.vhd`, and `mvlutil.vhd` files from the logiblox library:

```
LIBRARY logiblox;
USE logiblox.mvlutil.ALL;
USE logiblox.mvlarith.ALL;
USE logiblox.logiblox.ALL;
```

This has to be changed since you are trying to use the libraries that you already compiled in your work directory. Thus the above must be changed to:

```
--LIBRARY logiblox;
USE work.mvlutil.ALL;
USE work.mvlarith.ALL;
USE work.logiblox.ALL;
```

Now the libraries that will be used are compatible with the latest version of the simulation compiler and can be used in interpreted mode.

A sample script file that compiles all design components is shown below:

```
vhdlan -i mvlutil.vhd
vhdlan -i mvlarith.vhd
vhdlan -i logiblox.vhd
vhdlan -i dpmem.vhd
vhdlan -i reg.vhd
vhdlan -i reg_le.vhd
vhdlan -i mult2k4.vhd
vhdlan -i madd4.vhd
vhdlan -i const.vhd
vhdlan -i mult.vhd
vhdlan -i add2.vhd
vhdlan -i add3.vhd
vhdlan -i bit_add.vhd
vhdlan -i SWITCH1.vhd
vhdlan -i SWITCH2.vhd
vhdlan -i SWITCH3.vhd
vhdlan -i SWITCH4.vhd
vhdlan -i SWITCH5.vhd
vhdlan -i DOUBLE_fsm.vhd
vhdlan -i ADD_fsm.vhd
vhdlan -i add_array_lower.vhd
vhdlan -i reg_bank.vhd
vhdlan -i add_bank.vhd
vhdlan -i HYBMULT.vhd
vhdlan -i IO_fsm.vhd
vhdlan -i system.vhd
vhdlan -i TB_SYSTEM.vhd
```

The order in which the files are compiled does matter and the files in the lowest level of design hierarchy tree as well as all library components must be compiled first. Performing this compilation requires that you have a working directory (usualy named WORK) in the simulation directory as well as the setup filed descibed previously. The `vhdlan` compiler checks only for syntax errors. In other words, your design may compile without errors but the simulation waveform will show incorrect functionality. Sometimes, the error may be very subtle or hard to find even when using the waveform simulator and debugger. At this point it may be necessary to perform synthesis just to find out the source of the error.

2.5 Waveform Analisys and Debbuging

Once the source files are sucessfully compiled, the debugger can be invoked by typing `vhdldbx &`. This tool is a graphical debugging and waveform analysis application. The first step is to choose the configuration primitive associated with the test-bench. As shown in Figure 2.2, in our example the configuration file is `CFG_TB`. The configu-

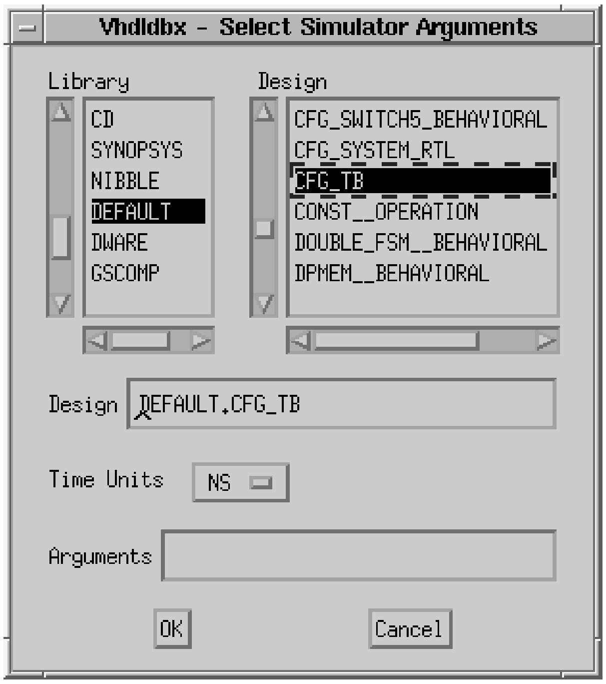

Figure 2.2: VHDLDBX start menu

ration mapping is usualy defined at the end of the test-bench file as shown below:

```
configuration CFG_TB of SYSTEM_TESTBENCH is
```

```
        for TEST
                for UUT : system
                        use configuration WORK.CFG_SYSTEM_RTL;
                end for;
        end for;
end CFG_TB;
```

The above definition ensures that a correct configuration file is used for the component instatiated in the test-bench file (top level file). For more information on configuration definition and its uses, please refer to the SOLD manual.

Once the configuration is picked, the main debugger window is invoked. This window is shown in Figure 2.3. This window shows all your source code and signals that are implented. You can place break points in all reasonable places in the code and step through the VHDL processes. It is also possible to write script files for the debugger and run then in this window. This feature is quite usefull during pos-place-and-route verification stage as the signals become buried and hard to find graphically.

The waveform viewer can be invoked with the `trace signal_name` arguement in the debugger or through the hierarchy browser shown in Figure 2.4. This tool is invoked from the `Misc` menu in the debugger. The hierarchy browser proved to be quite usefull when searching for signals buried very deep in the design. The speed of this tool however is not very impressive for large designs simulated after place-and-route stage. One reason for this is because everything is compiled in interpreted mode. Thus using the compiled mode and the C Developers Bundle for HPs should speed-up the simulation process.

The traced signals can be viewed and the test-bench results can be compared with other design models. This is the last step of this stage after which the designer should be confident that the design functions logicaly. This step does not ensure that final product will work as the timing characteristics can only be verified after the place-and-route method.

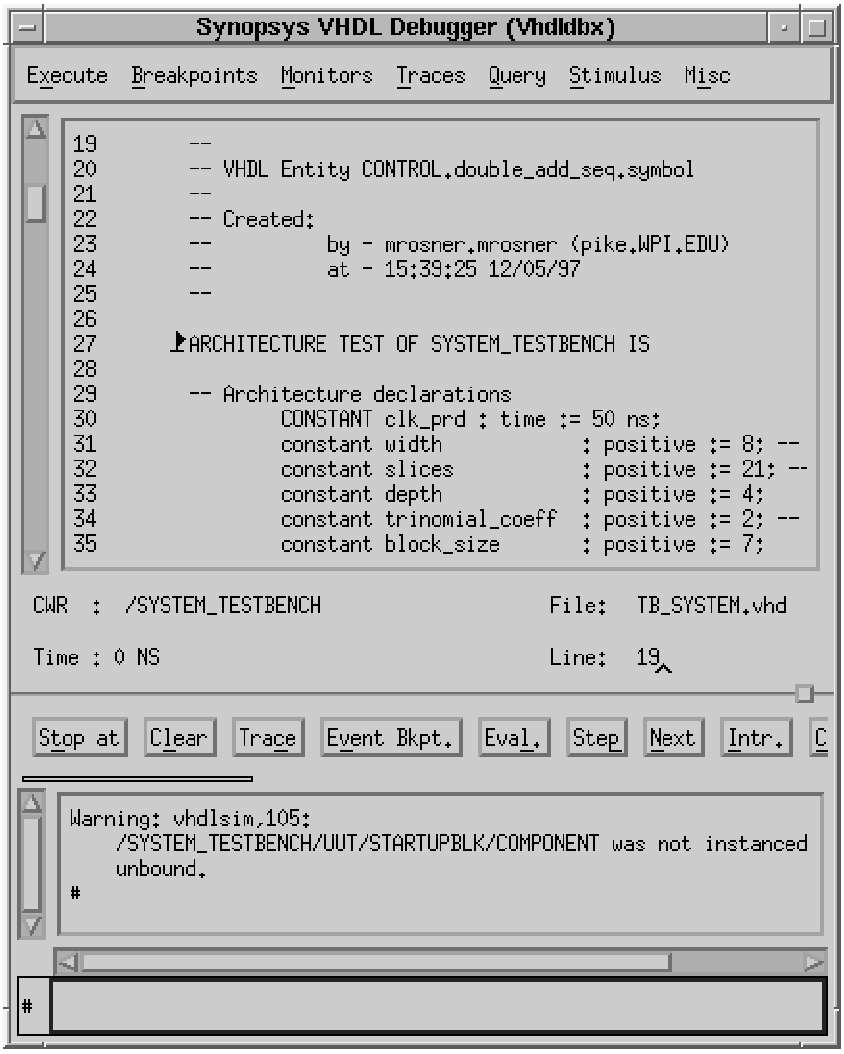

Figure 2.3: VHDLDBX debugger

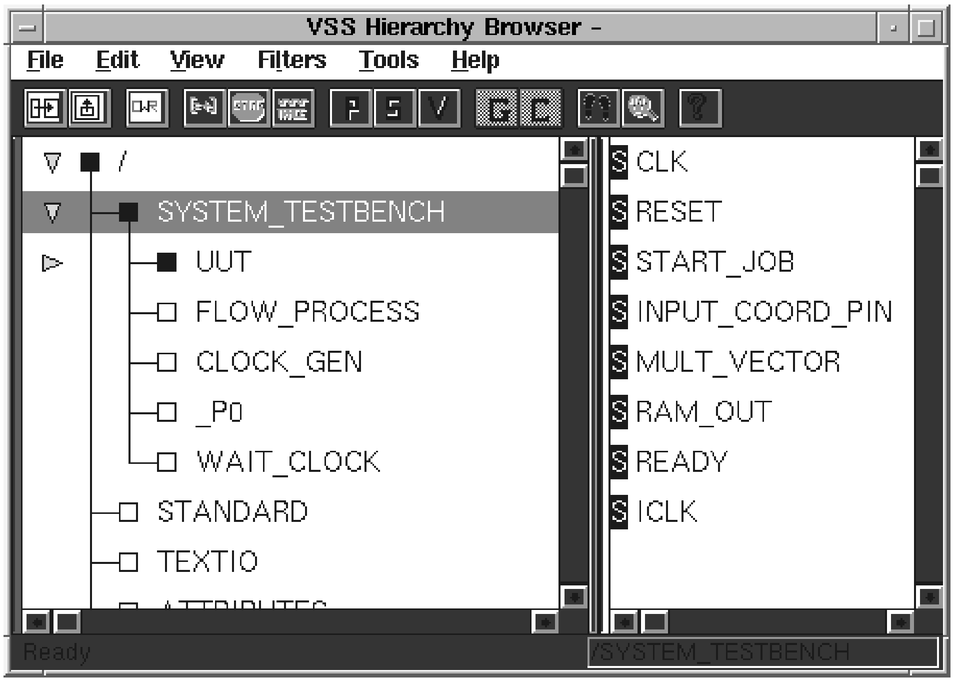

Figure 2.4: VHDLDBX hierarchy browser

Chapter 3

Synthesis

In this section I will try to introduce the synthesis process done with the Synopsis tools. This is only an introduction to the entire synthesis process. Naturally, synthesis will differ from one design to another and this chapter shows only the common aspects that may become important. The reader is urged to extensively use the SOLD manual.

3.1 Setup Conditions

All setup conditions are defined in the file `.synopsys_dc.setup`. In this file you should define device libraries and paths to the library files, path to the working directory, path to design_ware libraries, and few other settings such as cache location.

```
/* -------------------------------------------------------------- */
/* FPGA COMPILER STARTUP FILE - .synopsys_dc.setup                */
/* FOR XC4062xl PARTYPES using Xilinx M1.3                        */
/* Elliptic Curve Implementation                                  */
/* Martin C. Rosner                                       2/17/98 */
/* -------------------------------------------------------------- */
        XilinxInstall = get_unix_variable(XILINX);
        SynopsysInstall = get_unix_variable(SYNOPSYS);
        search_path = { .  \
                        XilinxInstall + /synopsys/libraries/syn \
                        SynopsysInstall + /libraries/syn }
/* -------------------------------------------------------------- */
/* Define a work library. You must create the subdirectory 'work'  */
```

```
/* ------------------------------------------------------------- */
        define_design_lib WORK -path ./WORK
/* ------------------------------------------------------------- */
/* Declare the use of the Xilinx DesignWare library for use during */
/* synthesis. Here I select the XC4000E DesignWare library        */
/* ------------------------------------------------------------- */
        define_design_lib xdw_4000xl \
            -path XilinxInstall + /synopsys/libraries/dw/lib/xc4000xl
/* ------------------------------------------------------------- */
/* General configuration settings                                */
/* ------------------------------------------------------------- */
compile_fix_multiple_port_nets = true
xnfout_constraints_per_endpoint = 0
xnfout_library_version = "2.0.0"
bus_naming_style = "%s<%d>"
bus_dimension_separator_style = "><"
bus_inference_style = "%s<%d>"
cache_read = /local2/martin
cache_write = /local2/martin
xlnx_hier_blknm = 1
/* ------------------------------------------------------------- */
/* output of synlibs -fc 4062xl-1                                */
/* ------------------------------------------------------------- */
link_library = {xprim_4062xl-1.db xprim_4000xl-1.db xgen_4000xl.db xfpga_4000xl-1.db xio_4000xl-1.db}
target_library = {xprim_4062xl-1.db xprim_4000xl-1.db xgen_4000xl.db xfpga_4000xl-1.db xio_4000xl-1.db}
symbol_library = {xc4000ex.sdb}
synthetic_library = {xdw_4000xl.sldb standard.sldb}
```

The libraries defined at the bottom of the setup file (link_library, target_library, symbol_library, synthetic_library) are device specific. Consequently this file has to be updated every time you decide to compile/synthesize for a different device. Not to worry, as there is an easy way to do that. By using the `synlibs` command you can find all libraries necessary for a specific device. To list all available devices, type `synlibs -fc -h`. Hopefully the device you need is there. Then type `synlibs -fc part_name` to get the listing of all necessary libraries. You can cut this result and paste it right into the setup file.

3.2 FPGA Analyzer

The FPGA analyzer is a graphical tool that we used to perform synthesis on our designs. The main window shown in Figure 3.1 is invoked with `fpga_analyzer` command. In this window you can view the compiled design, perform timing and area analysis, and run the synthesis process. The FPGA analyzer also includes state-machine features that were not explored to their full potential during the course of our project. You can use the state-machine extraction features to isolate the state-machine from the entire design and set attributes such as encoding to the state machine. These features are described in SOLD manual. From the main FPGA analyzer

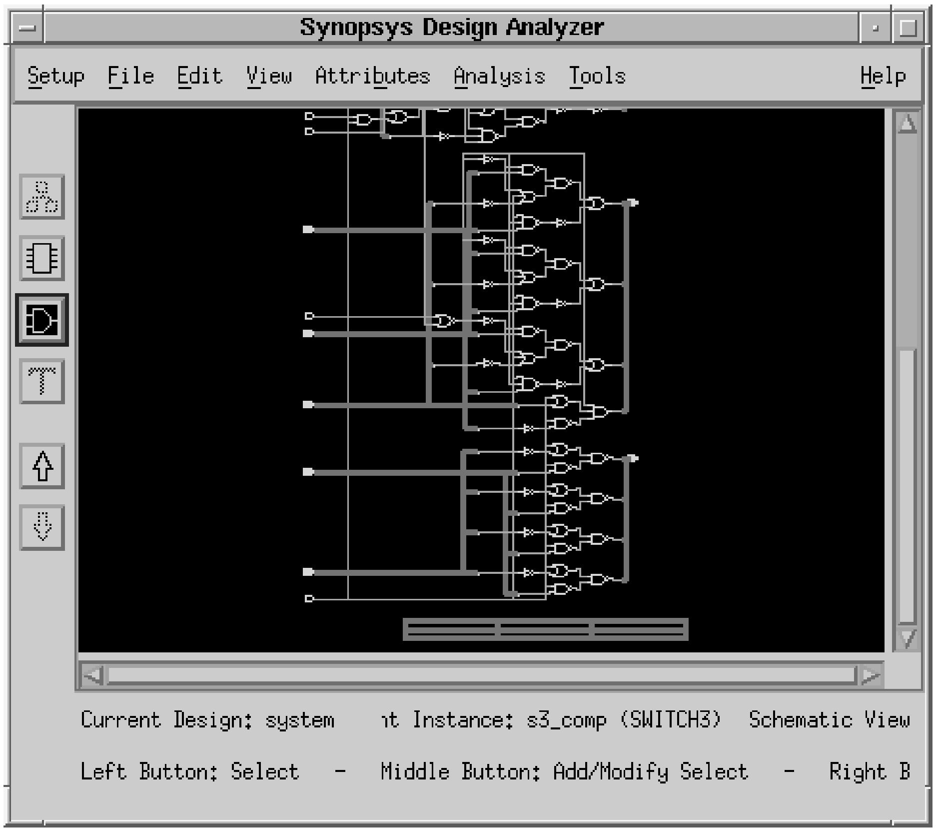

Figure 3.1: Main FPGA analyzer window

window, you can also invoke a command window and issue all instructions through a command line. After a while, this option proves to be faster because you can run numerous script files that get executed much faster. Eventually, we performed the entire synthesis by executing script files. This enabled us to invoke many synthesis sessions resulting in faster turnaround cycle.

3.2.1 Synthesis Script

The script shown below is an example of the entire synthesis process that was used to compile the elliptic curve engine.

Component List

```
/* Sample Script for Synopsys to Xilinx Using */
/* FPGA Compiler targeting a XC4000EX device */ /* Set the name of the design"s top-level */
        TOP = system
        F1 = packages
        F2 = dpmem
F3 = reg
F4 = reg_le

F5 = add2
F6 = add3
F7 = bit_add
        F8 = mult2k4
        F9 = madd4
        F10 = const

F11 = mult
F12 = DOUBLE_fsm
F13 = ADD_fsm
        F14 = IO_fsm
F15 = reg_bank
F16 = SWITCH1
F17 = SWITCH2
F18 = SWITCH3
F19 = SWITCH4
F20 = SWITCH5
F21 = add_array_lower
F22 = add_bank
F23 = HYBMULT

        designer = "Martin Rosner"
        company  = "WPI Crypto Group"
        part     = "4062XLPG475-1"
```

In this part you can define all variables that you will use throughout the script including all components and the top-level design name which, in this case, is system. Here you can also define the part/device that you are compiling for and the designer and company name.

Reading Design Components

Before you can compile the design, it has to be read into the analyzer. Some files may already be in acceptable format in which case you only need to read them. This is the case with the *.edn files that represent the LogiBloX components. These components are read in from the working directory. The *.edn files are created by the LogiBloX generator by producing the gate-level EDIF netlist as shown in Figure 2.1. The created *.edn files have to be placed in the working directory in order for the synthesis process to be able to read the LogiBloX components.

```
/* Analyze and Elaborate the design file.       */
        analyze -format vhdl "vhdl/" + F1 + ".vhd"

        read -format edif "WORK/" + F2 + ".edn"
        read -format edif "WORK/" + F3 + ".edn"
        read -format edif "WORK/" + F4 + ".edn"

        analyze -format vhdl "vhdl/" + F5 + ".vhd"
        analyze -format vhdl "vhdl/" + F6 + ".vhd"
        analyze -format vhdl "vhdl/" + F7 + ".vhd"
        analyze -format vhdl "vhdl/" + F8 + ".vhd"
        analyze -format vhdl "vhdl/" + F9 + ".vhd"
        analyze -format vhdl "vhdl/" + F10 + ".vhd"
        analyze -format vhdl "vhdl/" + F11 + ".vhd"
        analyze -format vhdl "vhdl/" + F12 + ".vhd"
        analyze -format vhdl "vhdl/" + F13 + ".vhd"
        analyze -format vhdl "vhdl/" + F14 + ".vhd"
        analyze -format vhdl "vhdl/" + F15 + ".vhd"
        analyze -format vhdl "vhdl/" + F16 + ".vhd"
        analyze -format vhdl "vhdl/" + F17 + ".vhd"
        analyze -format vhdl "vhdl/" + F18 + ".vhd"
        analyze -format vhdl "vhdl/" + F19 + ".vhd"
        analyze -format vhdl "vhdl/" + F20 + ".vhd"
        analyze -format vhdl "vhdl/" + F21 + ".vhd"
        analyze -format vhdl "vhdl/" + F22 + ".vhd"
        analyze -format vhdl "vhdl/" + F23 + ".vhd"
        analyze -format vhdl "vhdl/" + TOP + ".vhd"
        elaborate TOP
```

The rest of the component have come in the VHDL form. These pieces have to be analyzed to check for syntax and major design errors (such as multiple nets connected error). The order in which the files are analyzed does matter and elements in the lowest level of the design hierarchy have to be analyzer first. Consequently the top/system level file is analyzed last.

LogiBloX Issues

Since LogiBloX components are proprietary to XILINX, it is not necessary to synthesize them and the Synopsys tools are not needed for these components. In cases where the LogiBloX component is inside a larger design, you must set the `dont_touch` attribute on those blocks containing LogiBloX. The portion of the script shown below, identifies two modules with LogiBloX components and sets the `dont_touch` attribute on three distinct LogiBloX.

```
/* Set the current design to the reg_bank level.        */
        current_design F15
/* Don't touch the logiblox*/
        set_dont_touch (bank_i_*)

/* Set the current design to the HYBMULT level.         */
        current_design F23
/* Don't touch the logiblox*/
        set_dont_touch (BANK_REG_LE_*)
        set_dont_touch (BANK_REG_*)
```

First LogiBloX in contained inside component F15 which is the `reg_bank` module. Within the `reg_bank` module all instances of `bank_i_*` LogiBloX are identified with the `dont_touch` attribute. Similarly, two more LogiBloX instances are treated inside module F23 which is the hybrid multiplier.

By setting the `dont_touch` attribute on LogiBloX, the synthesis compiler will 'skip' these components when performing logic optimization and synthesis. The *.edn file that was placed in the working directory provides Synopsys with information that is used to estimate net delay and area utilization through the LogiBloX component. This however is only an estimate since the final results for a specific LogiBloX or macro is not available until after the place-and-route stage. From our experience, we have seen that area estimate given by Synopsys for a design that contains LogiBloX is quite accurate (within 10% of actual). Timing results are heavily dependent on the availability of routing resources inside a particular device. Therefore, Synopsys approximation of timing delay and minimum clock frequency tend to be little too optimistic when LogiBloX components are included. In any case, reports generated by Synopsys provide a very good rough estimate of speed and area performance that can be used by the designer to identify critical paths before committing the design to the next stage which tends to be very time consuming.

Setting Initial Constraints

Setting design constraints is an important part of the synthesis process and should not be omitted at this point. Few constraints shown here are very basic and the reader is encouraged to look in other documentation such as the SOLD manual, to learn about the full capabilities of Synopsys constraints. Besides setting a clock period constraint, the designer can also put restrictions on paths that are known to result in long delays. By doing this, Synopsys tool will work extra hard to optimize all modules in that path in order to meet a path constraint. In addition to timing constraints, you can also put area restrictions on particular block of the design to further optimize the final netlist. Below is a brief example of setting some typical constrains in the script file.

```
/* Set the current design to the top level.        */
        current_design TOP
remove_constraint -all
/* Uniquify the design and reset the schematic */
        uniquify
        create_schematic -size infinite -gen_database

/* include timming and area contraints */
create_clock clk -period 100
set_input_delay 0 -clock clk { all_inputs()}
set_output_delay 0 -clock clk { all_outputs()}
set_operating_conditions WCCOM

/* Indicate which ports are pads.    */
        set_port_is_pad "*"

set_pad_type -no_clock all_inputs()
set_pad_type -clock clk
        set_pad_type -slewrate LOW all_outputs()
        insert_pads
```

Besides the clock period constrains, input pad delay can be defined. In this example these delays are 0ns which means that all inputs are delayed by 0ns with respect to the clock input. Wire-load information and delays are contained in files provided by XILINX and are invoked by setting the operating conditions. In this example, we set worst-case-commercial (WCCOM) conditions as applied to the specific XILINX device. The last step of this script excerpt inserts virtual pads in the design so that Synopsys has information regarding input and output pad delays.

Compiling and Generating Reports

To start compilation you can run the `compile` command along with few options as shown below.

```
/* Synthesize the design.*/
/*        compile -map_effort high -ungroup_all*/
    compile -boundary_optimization -map_effort high
```

All of these options and more are explained in the SOLD manual and perhaps the most important ones are `-map_effort` and `-boundary_optimization`. With boundary optimization turned on, the compiler with merge all components into one file and perform optimization across all boundaries inside the top-level design. Synthesis will take longer with this option turned on, however it may result in better performance as well. If the designer knows exactly how to partition the design, boundary optimization may be turned off. In this mode, Synopsys will optimize all individual components of the design separately and then merge the results to generate the netlist. As long as the critical path does not span over boundaries, turning off boundary optimization may result in faster synthesis and better performance. The boundary optimization feature does not apply to LogiBloX as we've already place a `dont_touch` attribute on these blocks.

The following includes some shell commands that perform some cleanup of older reports, as well as the commands that generate new timing, area, and constraint reports.

```
/* Write the design report files.                */
        sh rm -f "reports/" + TOP + ".old"
        sh cat "reports/" + TOP + ".fpga" "reports/" + TOP + ".timing" "reports/" + TOP + ".cnst" > "reports/" + TOP + ".old"
        report_fpga > "reports/" + TOP + ".fpga"
        report_timing > "reports/" + TOP + ".timing"
        report_constraint -verbose > "reports/" + TOP + ".cnst"
```

The result of these actions are three files containing new reports in the report directory, as well as an old report that can be used to compare the results.

Synopsys synthesis is performed at the FPGA level. This means that the design is optimizes not at the gate level but at the CLB or logic block level. The advantage of such optimization is obvious as we are trying to fit the design into a device that contains only CLBs as digital elements. Thus, since Synopsys has detailed information about CLBs from libraries defined earlier, it can perform synthesis at this level. The design should be saved after compiling to preserve the CLB level netlist. This netlist can be used to extract state-machines, generate reports, and perform path and critical timing analysis. This netlist cannot be used however as an input to the XILINX tools. In order for XILINX to be compatible with many different synthesis tools, the vendor requires the netlist to be in gate level format. This can be obtained by running the `replace_fpga` command.

```
/* Write out an intermediate DB file to save state */
        write -format db -hierarchy -output "db/" + TOP + "_compiled.db"

/* Replace CLBs and IOBs primitives (XC4000E/EX/XL only)    */
        replace_fpga
```

The resulting design and the netlist are saved

```
/* Set the part type for the output netlist.      */
        set_attribute TOP "part" -type string part

/* Write out the intermediate DB file to save state*/
        write -format db -hierarchy -output "db/" + TOP + ".db"
```

and the netlist is used in the place-and-route stage.

```
/* Save design in XNF format as <design>.sxnf      */
        write -format xnf -hierarchy -output "sxnf/" + TOP + ".sxnf"
```

Additionally, all constraints are save in a *.ncf file and used by the place-and-route tools to optimize the design.

```
/* Write out the timing constraints               */
        ungroup -all -flatten
        write_script > "dc/" + TOP + ".dc"

/* XILINX primitive to convert Synopsys design constraints to Xilinx format*/
sh dc2ncf "dc/" + TOP + ".dc"
```

3.3 XILINX Specific Issues

XILINX FPGA devices include many features that can be used to improve the performance of the design. Most of the information about such features can be found in [7]. Information on how to utilize these features can be found in SOLD manual, the *HDL Synthesis For FPGAs* manual available in the Crypto lab and the WWW, and XILINX and Synopsys WWW help pages. The two most common features that are used to improve performance are the STARTUP block and global nets inside the XILINX FPGA. These are briefly described below.

3.3.1 Startup Block

The XC4000 device has a dedicated set/reset net that is routed to every flip-flop in every CLB in the device. Thus you do not need to utilize general purpose nets to route the reset signal to each flip-flop. This ensures fast path to all CLBs and frees additional routing resources that can be used for the rest of the design. The STARTUP block is a component defined in the XILINX library path that can be instantiated in the same way as any module inside your design. Thus you first declare the component

```
component STARTUP
                PORT ( GSR : IN std_logic );
end component;
```

and then instantiate it inside the architecture body.

```
STARTUPBLK: STARTUP port map (GSR => reset_internal);
```

Your global reset signal is also fed to the GSR input of the STARTUP block. This ensures that the dedicated set/reset net is utilized during the place-and-route stage. The STARTUP block is defined only for the synthesis tools. This means that when you perform RTL simulation, you will receive a warning saying that the simulation compiler could not infer the STARTUP block. This will not change RTL simulation because the GSR is an input to the STARTUP block.

3.3.2 Internal Global Buffers

The 4000 series devices also contain dedicated routing resources that can be used for high fanout signals. By using these resources you will ensure the smallest possible net delay for the high fanout signal. This is generally advantageous when the high fanout signal is in the critical path. Once again, a special component has to be instantiated in the VHDL code to ensure that the global routing resources are utilized. Example below shows such instantiation of BUFGS_F which is a XILINX library component.

```
--declare component
component BUFGS_F
              PORT ( I : IN std_logic;
                     O : OUT std_logic);
end component;

--instantiate component
FORCEBUF: BUFGS_F port map (I => in_clk_en,
                            O => clk_enable_hf);
```

Usually a clock enable signal exhibits high fanout. In the example above the clock enable signal `in_clk_en` is passed through the buffer BUFGS_F. The resulting clock enable line `clk_enable_hf` is the signal that should be used for a high fanout load.

During RTL simulation, the BUFGS_F instantiation will cause a problem. That's because this component is not known to the simulation tool and the component drives many other modules. Thus, to perform RTL simulation, you need to comment out the BUFGS_F instantiation and connect `in_clk_en` directly to `clk_enable_hf` (`clk_enable_hf<=in_clk_en`).

Chapter 4

Place and Route

The place-and-route process was originally performed on the XILINX tools installed on the HPs. This was a convenience because all tools up to this point have been running on HPs as well. The HPs however proved to be very slow and we had to migrate this process to the PC. Even running on a 200MHz Pentium with 64M of RAM, the compilation required days and even weeks. Needless to say this process is the most time consuming one and the designer should be aware that large designs will take quite a long time. The designer should be fairly confident of the correct functionality of his/hers module before committing to this stage of the process.

4.1 Setup

The M1 tools were run on PC and after installation minimal setup was required. You should create a directory on the C drive (working over the network slows down the process) and copy the netlist files generated by Synopsys into that directory. Xilinx tools will keep different revisions of the place-and-route results in separate folders automatically.

4.2 Input Files

Xilinx tools can read *.edn and *.sxnf netlist formats. Since Synopsys creates the *.sxnf file you will need to use that as input to the Xilinx tools. In addition, if you created a constraint file with Synopsys, you should use it in the place-and-route stage as well. This is the *.ncf or netlist constraint file, that contains clock, net, area, and pad constraints. In addition to the netlist constraint file you can specify your own constraints in the user constraint file (*.ucf) and also use it as input to the Xilinx tools. The *.ucf file takes priority over the *.ncf file so you can use it to overwrite the constraints that you have set during synthesis.

4.3 Placing and Routing Your Design

The main tool used during this process is the Foundation Series software M1.3.7. This was the latest version of XILINX place-and-route tools available for their devices. Figure 4.1 shows the main design manager window. The design manager allows you to compile many different versions of the design. Thus you can compile for different devices or with different timing constraints and different mapping and routing efforts. You may want to try initial place-and-route process using relaxed compiler settings. This will speed up the process and allow you to do post place-and-route verification of the design rather quickly during the early stages of this procedure. Once the design has been verified after the place-and-route process, you should run a final place-and-route with maximum efforts and better timing constrains. We have found that running through many iterations and tightening the clock constraint improves the performance of the design up to a certain limit.

There are quite a few options that you may choose before starting the place-and-route process. All of these options are explained in the Xilinx manual available from the ECE department. In addition you may reference the *HDL Synthesis For FPGAs* manual available from Xilinx WWW or in the Crypto Lab. If you are using the netlist

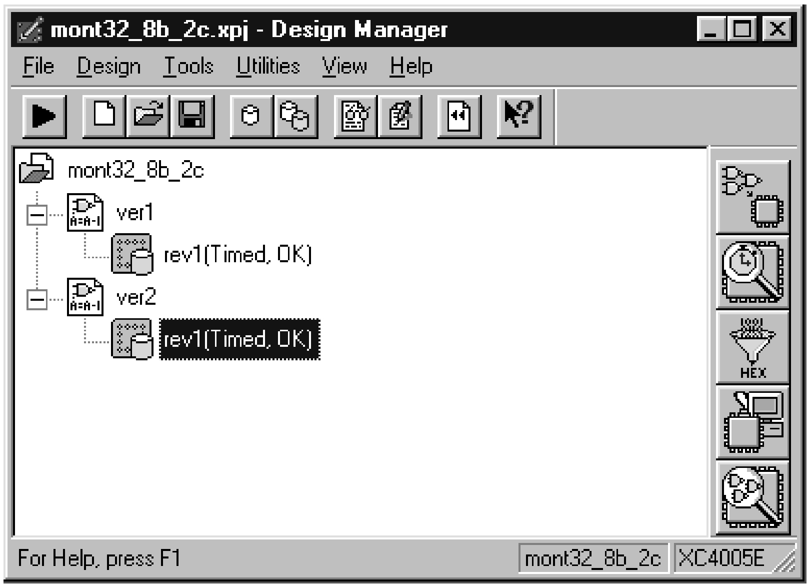

Figure 4.1: Xilinx design manager

constraint file generated by Synopsys, You will need to delete/comment out one line in that file. The file that reads

```
NET "clk" OFFSET = IN : 0 : AFTER : "clk";
```

should be changed to

```
#NET "clk" OFFSET = IN : 0 : AFTER : "clk";.
```

For some strange reason Synopsis creates an OFFSET definition for the clock with respect to the clock. Xilinx does not like that definition, and since it is not needed anyway, you should delete it. Figure 4.2 shows the menu where you can define different options, and constrains, and guide files. In order to do the post place-and-route verification using Synopsys, you must create a simulation file in the VHDL format as shown in Figure 4.2. In the next chapter I will describe how to use this VHDL file to verify your final design. Some other options that you should pay attention to are the optimization and mapping settings, as well as the place and route configuration.

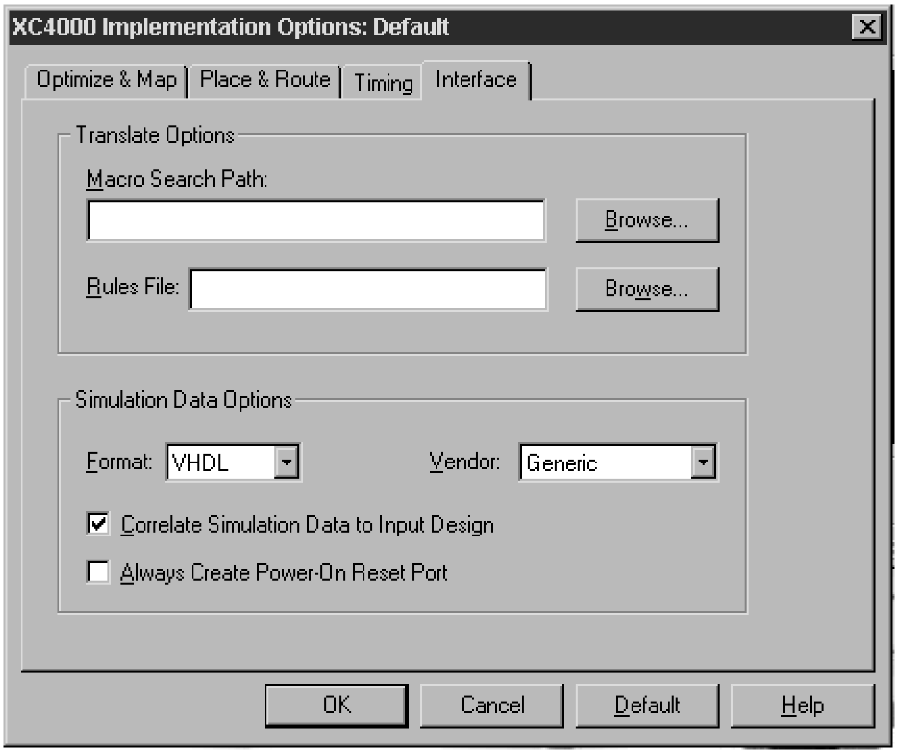

Figure 4.2: Implementation options menu

Figure 4.3 shows the flow engine that is invoked during the implementation stage of the place-and-route process. This window will update you on the progress of the compilation and show you any errors and warnings that may result in the different stages of the process. Some of the warnings that may come up should not cause great concern. Xilinx M1 posts warning when it finds `logic_0` or `logic_1` connections generated by Synopsis. These are just power and ground connections and the M1 tools know what to do with them. The M1 tools also generate warnings when signals are merged and blocks are optimized. This is normal. When all stages of the compilation are completed without errors, the simulation VHDL file (*time_sim.vhd*), as well as the timing characteristics file (*time_sim.sdf*) are generated.

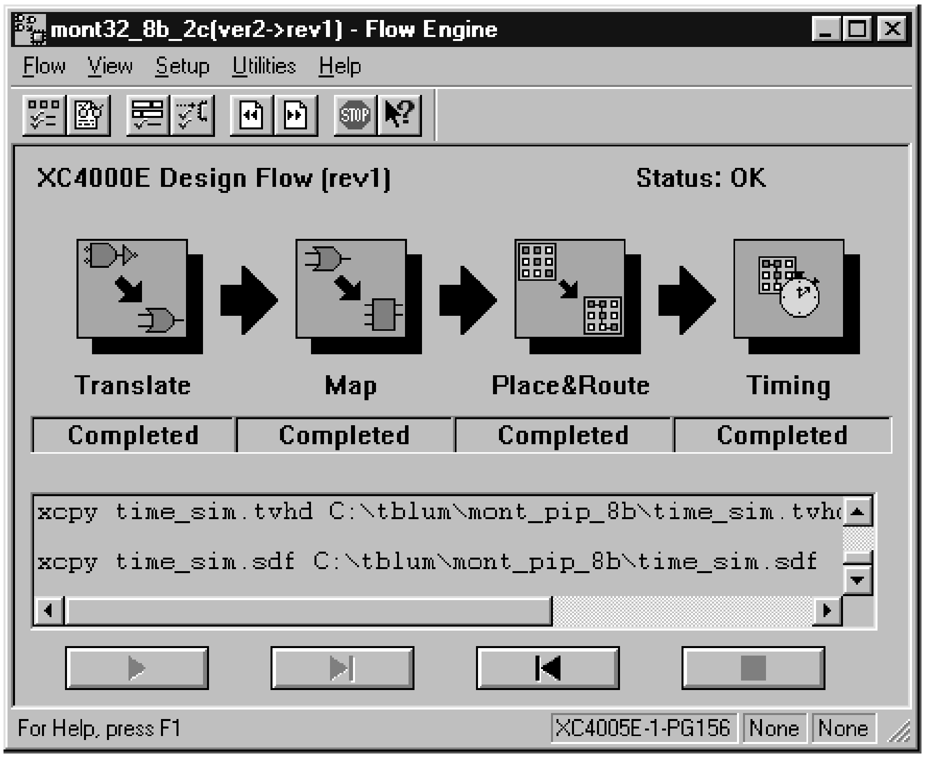

Figure 4.3: Xilinx flow engine

Chapter 5

Post Place and Route Verification

The final step of the design process involves verification of your system after the place-and-route stage. Correct operation of the design can now be verified in the same manner as the register transfer level verification. During this stage, simulation is performed based on the timing information provided by the Xilinx place-and-route tools. This information includes the net, CLB, and pad delay that were incurred during the place-and-route stage.

5.1 Setup

The setup file `.synopsys_vss.setup` stays unchanged except for the definition of the `TIMEBASE` variable as shown below.

```
TIMEBASE                = ps
TIME_RES_FACTOR         = 1
NO_HAZARD_MESG          = TRUE

WORK            > DEFAULT
DEFAULT         : ./WORK
```

```
CFLAGS  = -Dhp700Architecture
LDFLAGS = /bin/ld -b -o %s
```

The post place-and-route simulation is more precise because all delays are calculation to the nearest pico-second. For this reason, the `TIMEBASE` variable has to be changed to pico-seconds as well. This also means that the post place-and-route simulation will take longer to compile since it has to calculate 1000 points for every point calculated during the RTL simulation.

5.2 Input Files

The two files generated by Xilinx M1 tools are `time_sim.sdf`, and `time_sim.vhd`. You will need these files in you simulation directory. The `time_sim.sdf` file contains all timing information that resulted from a specific place-and-route session. The `time_sim.vhd` is the simulation file that contains your design. This VHDL code looks nothing like the code you wrote because it is defined in terms of Xilinx CLBs and net connections. This means that it will be very hard and sometimes impossible to find the signals that you want to trace during this simulation.

In addition to your design files, some library components are needed. These are `simprim_VITAL.vhd`, `simprim_Vcomponents.vhd`, and `simprim_Vpackage.vhd`. These files contain library definitions of all function block found inside the Xilinx device and are called by your design file. The library files can be found in the Xilinx directory `$XILINX/vhdl/src/simprims/`. Normaly, these library files are compiled only once during instalation or upgarde of Synopsys tools and the compiled files would be accessed through the `SIMPRIM` library. This is not true in our case. The reason why we had to compile the simulation libraries every time is that we didn't have the appropriate C compiler on the HPs at that time. Consequently the libraries could not be generated with the following commands:

```
vhdlan -c -optimize -w SIMPRIM  $XILINX/vhdl/src/simprims/simprim_Vcomponents.vhd
vhdlan -c -optimize -w SIMPRIM  $XILINX/vhdl/src/simprims/simprim_Vpackage.vhd
vhdlan -c -optimize -w SIMPRIM  $XILINX/vhdl/src/simprims/simprim_VITAL.vhd.
```

Once the appropriate compiler is available, all necessary simulation libraries, including LogiBloX libraries, should be recompiled and stored permanently in the Xilinx directory.

5.3 System Test-bench File

The test-bench file that you used during the RTL simulation can also be used at this stage. However, it is important to keep in mind that the post place-and-route simulation file `time_sim.vhd` may have different port definitions. This may occur when you have array structures or variables as inputs or outputs in your original VHDL code. After the place-and-route process variables, and arrays are changed to physical ports (signals) and thus the instantiation of your system in your system test-bench will no match the port declaration in the `time_sim.vhd` file. This is a minor change that you have to make.

5.4 Scripts

During the RTL verification we only used one script to compile the simulation files. In this stage we use another script to perform the actual simulation. This is done because, as mentioned previously, the simulation tools, especially the hierarchy browser, are very slow during post place-and-route simulation. Using a simulation script will speed-up the process as you do not have to take the time to traverse the hierarchy tree to find your signals. This will however require some initial research into the names of the signals that you want to see on the waveform viewer.

5.4.1 Compilation Script

Since, the number of design files is reduced to one, the compilation script for this simulation is much shorter even though the time required to compile these files may be longer. Thus, until the appropriate C compiler is available on HPs, you can compile all necessary files in interpreted mode as follows:

```
vhdlan -i simprim_Vcomponents.vhd \
          simprim_Vpackage.vhd \
          simprim_VITAL.vhd \
          time_sim.vhd \
          packages.vhd \
          TB_SYSTEM.vhd
```

After all files are compiled into the working directory you have to invoke the simulator with the following command line:

```
vhdldbx -t ps -sdf_top SYSTEM_TESTBENCH/UUT -sdf time_sim.sdf CFG_TB &
```

The optional flags defined by the command line ensure the correct time-base and define the *.sdf file that contains timing information. This definition is crucial to performing correct post place-and-route simulation. The command line will invoke the debugger/simulator, at which point you can use it in the same way as described in Chapter 2.

5.4.2 Simulation Script

The script file can contain all commands that you would normally use in the simulation debugger window. The most often used commands include `trace, evaluate, cd, step, next` and many more. For a full listing of commands you may want to invoke the help window from the simulation debugger. In any case, the trickiest part of post place-and-route simulation is trying to simulate the signals that you want. This is because your design file `time_sim.vhd` is rather large and contains signals for all nets in the routed design. On top of that, Xilinx M1 tools may have renamed some signal names, at which point you will have to refer to the Xilinx floor-planner to find the names of the nets that you want to simulate. In most part, the Xilinx tools are not so cruel and leave part of your signal and port names in the resulting net name. The best way to search for your signal names is to do an initial text search on the `time_sim.vhd` file. This should give you some idea as to the naming convention and allow you to find the signals in the floor-planner. After finding the signals, it is a matter of tracing them in the script file. When doing so, you should include the full path name through the hierarchy including the test-bench. For example, to trace signal `a_op_s1(8,3)` from your design you need to use the following command:

```
trace /system_testbench/uut/a_op_s1(8,3).
```

Or you can change into the design directory, `cd /system_testbench/uut`, and then trace only the signal name.

Comparing these simulation results to RTL simulation and a C model of the design finalizes this stage. At this point you are ready to implement you design on a piece of hardware by down-loading a file generated by the Xilinx M1 tools into an EPROM that can be later used to program the Xilinx device on start-up. More details on how that procedure should follow in the next chapter. If your design does not meet the specs, or does not work properly, you will need to turn back to Chapter 2 and perform the entire cycle again. Good Luck!

Chapter 6

Implementation

.......................?

Bibliography

[1] P. Alfke and B. New. *Implementing State Machines in LCA Devices.* Xilinx, Inc., San Jose, CA. Xilinx Application Note XAPP 027.001.

[2] M. Gschwind and V. Salapura. A vhdl methodology for fpgas. In Will Moore and Wayne Luk, editors, *Proceedings of the 5th International Workshop on Field Programmable Logic and Applications, FPL1995*, pages 208–217, Berlin, August/September 1995. Springer-Verlag.

[3] S. Hauck, G. Borriello, S. Burns, and C. Ebeling. Montage: An fpga for synchronous and asynchronous circuits. In *2nd International Workshop on Field-Programmable Gate Arrays*, Vienna, August 1992.

[4] S. Hauck, S. Burns, G. Borriello, and C. Ebeling. An fpga for implementing asynchronous circuits. In *IEEE Design & Test of Computers*, volume 11, pages 60–69. IEEE, Fall 1994.

[5] Scott Hauck. Asynchronous design methodology: An overview. In *Proceedings of the IEEE*, volume 83, pages 69–93. IEEE, January 1995.

[6] Synopsys. *Finite State Machines - Application Note.* Synopsys, Inc., Mountain View, CA, April 1995.

[7] Xilinx, Inc., San Jose, CA. *The Programmable Logic Data Book*, 1996.