PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 38, NO. 2, PP. 147-174 (1994)

USING LOGIC SYNTHESIS TOOLS FOR TEXAS INSTRUMENTS FPGAs

Géza NEMESSZEGHY

Department of Electronic Technology
Faculty of Electrical Engineering
Technical University of Budapest
Phone.: (36–1) 271–2330
E–mail: nemes@ett.bme.hu
H–1521 Budapest, Hungary

Received: April 10, 1994

Abstract

High density PLDs (Programmable Logic Devices) and FPGAs (Field-Programmable Gate Arrays) are becoming more and more popular in the field of logic design. Their ultimate advantages — no NRE (Non-REcurring) costs, fast time-to-market, in-house design, etc. — are being combined with ever increasing speeds and densities. Up to now the traditional FPGA design technique has been schematics. But hardware complexity has outrun schematics with chips so complex that the graphical representation of the circuit shows only a web of connectivity, not the functionality of the design. For this reason more and more engineers are turning to Hardware Description Languages (HDL) for digital design. The prospect of using Logic Synthesis Tools is one of the main reasons which make HDLs attractive for designers. These tools take a behavioural, or other type of HDL description, and produce a technology specific netlist for an FPGA or for another type of ASIC. The effectiveness of the Logic Synthesis Tools is a key factor in deciding against or in favour of HDLs and synthesis. The synthesis powers of two programs were tested and compared using three sample designs. The meaning of FPGAs, HDLs and Logic Synthesis are explained in more detail in the first chapters of the article. The results of logic synthesis are in the second part. The source codes, command line arguments and batch (or script) files used are also given.

Keywords: ASIC (Application Specific Integrated Circuit), Core, Design Analyser, Design Compiler, Exemplar, FPGA (Field-Programmable Gate Array), Synopsys, Ti'x'press, VHDL compiler.

Tools used:	Synopsys V3.0	→ Design Analyser
	(Sun workstation)	Design Compiler
		VHDL Compiler
	Exemplar	→ Logic Synthesis System (Core V1.21)
	(PC)	Ti'x'press (Subset of Core)
	Texas Instruments	→ ALS
	Model Technology	→ V–System/Windows
Source Code:	VHDL	
Designs:	– 16 bit adder	
	– 4 bit cascadable counter	
	– 16 bit counter	

1. Introduction

For over 25 years the most popular way of logic design was schematics. But today new techniques are making themselves felt in the world of digital design. Hardware complexity has outrun schematics with chips so complex that the graphical representation of the circuit shows only a web of connectivity, not the functionality of the design. With increasing densities it is becoming more and more essential to be able to express the functionality of the design in a high level language, hiding the details of implementation. Another trend that is calling for new design methods is the need for technology independence. This can be necessary, for example, when migrating from one technology to another. Schematics must use the functional blocks (called macros) provided by the technology in question, making migration rather difficult.

For reasons mentioned above more and more engineers are turning to Hardware Description Languages (HDL) for digital design. Top-down, HDL-based system design is most useful in the development of large projects, where several designers or teams must work concurrently. HDLs provide structured development, so that after the major architectural decisions have been made, work can proceed on several subdesigns. HDLs are quite similar to high level programming languages, and they are replacing schematics for more or less the same reasons as high level languages are replacing assembly languages. If the design is expressed in a hardware description language, it must be translated into real logic. This process is called logic synthesis. The logic synthesis tools take the HDL code (or another technology independent format) and, using the technology libraries, create a netlist. It is important to be aware of the synthesis powers of these tools, as this determines the usability of HDLs.

2. VHDL

VHDL is one of just a few HDLs in widespread use today, and it is recognised as a standard HDL by the IEEE (IEEE Standard 1076, ratified in 1987). The United States Department of Defence, as part of his Very-High-Speed Integrated Circuit (VHSIC) program, developed VHSIC HDL (VHDL) in 1982 (MIL-STD-454L). VHDL offers three types of constructs for digital design:

- *The structural description* method expresses the design as an arrangement of interconnected components. This is actually writing a netlist on a higher level.

- *The data-flow description* method is similar to a register-transfer language. This method describes the function of the design by defining the flow of information from one input or register to another register or output.
- *The behavioural description* method describes the function of a hardware design in terms of circuit and signal responses to various stimuli. The hardware behaviour is described algorithmically without showing how it is structurally implemented.

All of these methods can be intermixed in a single design description. For detailed information on hardware description languages you can find a list of VHDL textbooks at the end of the document.

2.1 VHDL simulation

Working in HDL is very similar to computer programming: we tell the computer in a high level language what we expect our design to do. In digital design though, writing the source code is only the first step in a long design-flow, so it is important to be sure that our description is correct. It can cost a lot of precious development time to simulate the design only after logic synthesis.

For VHDL there are a number of simulators running both on PCs and workstations (Synopsys VHDL Simulator, VIEWlogic's VHDL reader etc.). I have used a PC based program, the V- System/Windows from Model Technology. This compact simulation system provides a full VHDL environment and supports the complete IEEE 1076-1987 standard. First we must translate the VHDL file and then we can start the simulation. It is also possible to simulate designs that use vendor specific functions, data types or operators. These functions are usually defined in packages. So before simulating our design we first must translate these packages into the simulator, and specify the correct library name in the source code.

3. FPGAs

Basically an FPGA is a Programmable Logic Device (PLD), though a lot of sources use the abbreviation PLD in a more restricted meaning, excluding FPGAs. PLDs, in turn, belong to the family of ASICs (Application specific ICs), but the popular usage of this term only refers to other types of ASICs like Gate Arrays, Standard Cell, etc. (For simplicity I will use the term ASIC in this meaning). FPGAs differ from the rest of PLDs in their architecture: Usually programmable logic blocks are distributed on the surface of the chip, from which you can build up your design. In addition, there are some sort of routing resources used to interconnect the logic cells. There are I/O blocks as well, to connect the interior of the device to the outside. FPGAs are manufactured with CMOS technology. There are two methods in use to store the configuration data on the chips: volatile Static RAM (SRAM) cells and antifuses. SRAM FPGAs are two chip solutions, as an additional element (EPROM or EEPROM) is needed to store the data when the system is off power. The word antifuse refers to the fact that it works just the opposite way than a fuse: it conducts when it is programmed. The antifuse is much smaller than the SRAM cell, so an FPGA of that type can have smaller logic cells and offer more abundant routing resources. The FPGA families from TI use antifuses for data storage. The logic cells are arranged in horizontal channels. The horizontal interconnection lines run between, the vertical ones run over the logic cell rows. The TPC12 has two types of logic blocks: a combinatorial and a sequential cell. Both block types are multiplexer based. The cells are relatively small. This can be an advantage, as with larger blocks a lot of logic can go to waste when you only need to implement a simple logic function in a cell.

The main advantage of FPGAs is that they are Field-Programmable: the chip is programmed after all the manufacturing steps have been completed. This means fast time to market and no NRE (Non-REcurring) costs. Their densities, though, are smaller than ASICs. As a conclusion we can say that FPGAs (PLDs) are cheaper for smaller volumes, and ASICs (Standard Cell, Full Custom, Gate Array, Uncommitted Logic Array, etc.) are more economical for larger volumes. This makes FPGAs ideal for ASIC prototyping. It is also possible to start to manufacture an application with FPGAs and switch to ASICs for higher volumes later.

The name of these devices implies that they resemble Gate Arrays. This is only true for FPGAs having very simple and small logic blocks. For most of the chips there is considerable difference in the architecture. This means that it is not so simple to migrate from FPGAs to Gate Arrays. The migration becomes easier, if we can describe a design in a technology independent format. This format can also be used to switch from one FPGA or PLD to another, or to try to fit your design into more chips, to see which one gives the best result.

The technology independent format can be a truth-table, a state table, Boolean equations or a hardware description language, like VHDL or Verilog. These source codes then must be translated to real logic, and mapped into the technology used. This procedure is called Logic Synthesis.

3.1 Different FPGA Design Flows

When the first PLDs appeared on the market, the whole design and programming process had to be done by hand. Today a high-density PLD is unimaginable and practically not usable without the design software, which takes the design in one or in several input forms and leads the designer through all steps that must be performed. The software provided by the manufacturer usually includes mapping, optimisation, place & route and programming. The design entry and simulation software is generally a third-party product.

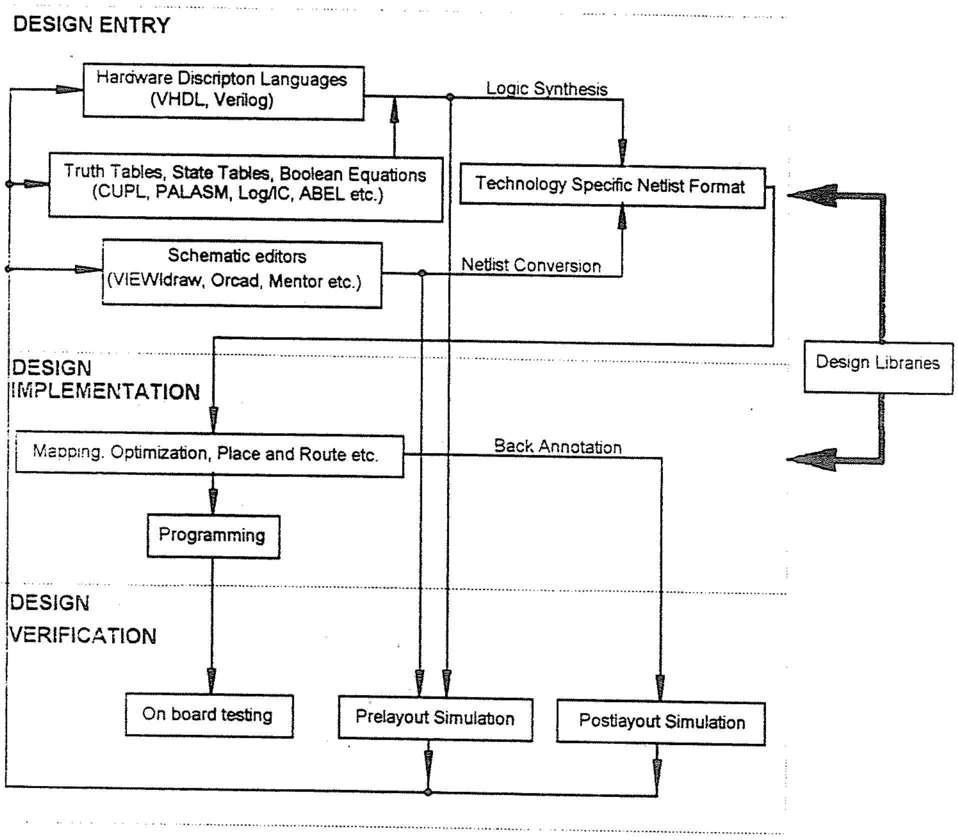
DESIGN ENTRY
Hardware Discripton Languages (VHDL, Verilog)
Logic Synthesis
Truth Tables, State Tables, Boolean Equations (CUPL, PALASM, Log/IC, ABEL etc.)
Technology Specific Netlist Format
Schematic editors (VIEWldraw, Orcad, Mentor etc.)
Netlist Conversion
DESIGN IMPLEMENTATION
Design Libraries
Mapping. Optimization, Place and Route etc.
Back Annotation
Programming
DESIGN VERIFICATION
On board testing
Prelayout Simulation
Postlayout Simulation

Fig. 1. FPGA design-flow

We can see the possible design flows of FPGAs (or PLDs) on Fig. 1. The first step must be the specification of the functionality of the circuit. This can be done basically in two different ways:

I. *Structural description:* Here we specify the Circuit's internal logical structure by describing the actual logical components and their interconnections. Again it can be done in more different ways:
   a) writing a netlist:
      – using a hardware description language that allows component instantiations (VHDL, Verilog);
      – using a technology specific or other netlist formats; (this would be rather tedious)
   b) using a schematic editor.

II. 'Functional' description: Now we are focusing more on what the circuit does rather than on its logical implementation. Designing this way is like writing a computer program. The name functional refers to the fact that here we are only concentrating on the functionality of the design. We can use the following tools:
   → hardware description languages;
   → state tables, diagrams;
   → truth tables;
   → Boolean equations.

There are a lot of languages that allow the use of state diagrams, truth tables and Boolean equations (CUPL, ABEL, Log/IC PALASM, etc.). HDLs represent a higher level of abstraction, and it is usually possible to mix structural and functional descriptions with them. The most prominent and generally accepted hardware description languages are Verilog and VHDL. The different entry formats must all be translated to a technology specific netlist format. For a structural description it is relatively easy: we only need a simple conversion program. Logic synthesis, that is translating a functional description, is a much more complex problem.

After Design Entry comes the Implementation and Verification phase. Usually we have to run several programs to map, optimise, place and route our design. Simulation is possible at more points of the design process. With prelayout simulation we can make sure that our design is functionally correct. Postlayout simulation takes place after place and route, when we already have the exact delay values. These values can be back annotated to the simulator.

4. Designing with Logic Synthesis Tools

After this general overview let us concentrate on that path of the design flow we are interested in. First a few things about the Synthesis Tools used. Synopsys is a complex design system running on workstations, and was originally created for ASIC design. Later support was added for FPGAs. The system has its own library editor, so in theory it can be used to design any FPGA, if all the necessary libraries have been written (Symbol Library, Technology Library). The software accepts a large variety of input formats, such as Verilog, VHDL, truth table, state table, Boolean equations, EDIF netlists, schematics, etc. A lot of parameters determine the way the synthesis works. You can choose from a set of optimisation strategies, and you can constrain your design. Constraining your design means that you tell the software what characteristics (speed, area, etc.) you expect to get after synthesis. The complexity of the system means that it is worth playing with these parameters trying to find the best result. But this is also a disadvantage, as trying out all the possibilities can take a lot of time, and sometimes it is really difficult to reproduce a result previously reached, but not saved. The basic design steps for VHDL code are as follows:

- Read in the VHDL source code;
- Constrain the design (set your goal);
- Compile (optimise) the design using the technology libraries.

There are a lot of other options, for example you can extract a state machine from the netlist and compile the state table format (you might reach a better result that way), or you can create a schematic using the symbol library. It should be noted that the output greatly depends on the input code. This means that if you are rerunning the optimisation phase over and over again with the same set of parameters, you are going to get different results every time.

Exemplar's *Complete Optimisation/Retargeting Environment (CORE)* was developed to allow the use of technology-independent design methods for FPGA, high density PLD and CMOS ASIC design. CORE accepts designs as equations, truth tables, netlists or VHDL descriptions, and it produces vendor specific netlist formats (Actel/TI, Xilinx, QuickLogic, Altera, LSI gate array). The software runs on PCs and on workstations as well. This system is simpler and easier to learn, but does not offer as much control of the synthesis as Synopsys does. Here you can use control files to constrain the design, and you can compile for minimum area or for maximum speed. I have synthesised VHDL code into ADL format (TI netlist), but it is also possible to optimise a technology specific netlist (e.g. generated by a schematic editor), or to retarget from a device to another. CORE also accepts EIL (Exemplar Integration Language) files, with which you can connect multiple designs in different formats into one large design.

TI'X'PRESS is also an Exemplar product, and it is a subset of CORE. It can only produce an ADL netlist from VHDL, PDS, PLA, OpenABEL, ADL and EIL files. The VHDL reader accepts a restricted version of VHDL, called Boolean 1076. In Boolean 1076 you can only use data-flow constructs. Behavioural description and component instantiation are not allowed. This means that if the VHDL source code is using a data-flow description, then both CORE and TI'X'PRESS should reach the same result.

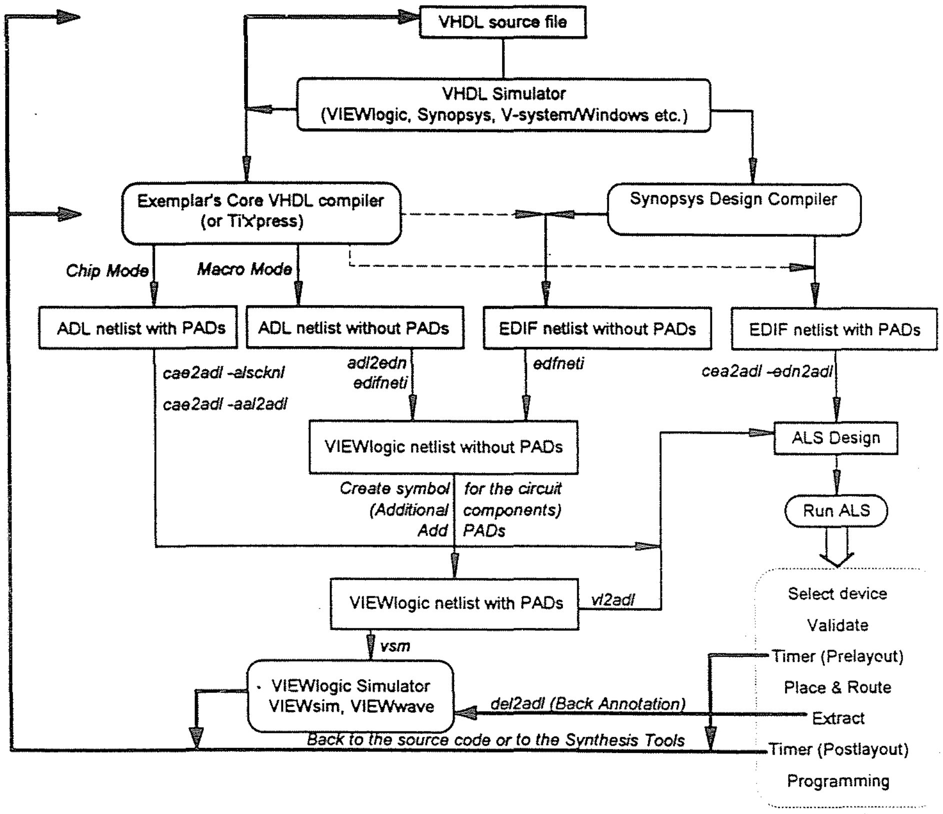

Fig. 2. Design-flow with Synopsys and CORE

Fig. 2 shows the design flow for TI TPC12 (10, 14) FPGA using Synopsys and CORE. After creating the VHDL file we should simulate it to see whether it works properly or not. When we are sure that the source code is correct, then comes the logic synthesis. Synopsys cannot generate an ADL file, so first we must make an EDIF netlist and then make the ALS Design with the ALS EDIF reader (cae2adl -edn2adl). The program can automatically insert the input and output buffers from the technology library, but the TPC12.db library does not specify any components as PADs. It includes all the PADs as simple components though, so they can be inserted in the VHDL source code by component instantiation. If the EDIF netlist does not have PADs, they can be added using a schematic editor supported by ALS. The most frequently used schematic editor with ALS is VIEWlogic's VIEWdraw. We must run VIEWlogic's EDIF reader (edifneti) to get a VIEWlogic (VL) netlist. After creating a symbol for the circuit we can add the PADs or other components, which can come from any other source (schematics, HDLs, truth tables, etc.). We can use the 'vl2adl' program to convert the VL netlist to an ALS Design.

With CORE it is more straightforward to make an ADL file directly from the VHDL source code. If you run the program in CHIP mode, then PADs are inserted on the ports. We can use the MACRO mode to prevent the placement of PADs. In MACRO mode we use our design as a component or subdesign. We can join these subdesigns with the EIL format or by converting it to VIEWlogic. For VL conversion we first must make an EDIF file (adl2edn) and then use 'edifneti'. CORE only makes a single ADL netlist, which is not enough to run ALS. You must create an ALS Design (which means the creation of additional files) by running 'cea2adl' with the '-alscknl' and the '- aal2adl' switches.

When we have an ALS Design we can run the 'als' program, which is a graphical environment. Within this environment you can perform several tasks (*Fig. 3*). First the device and package type must be selected. The next step is to run 'validate', which searchers for design errors (like missing PADs, fanout problems etc.). If your design has passed the validation, than you can run the Place & Route and the Extract functions. 'Extract' generates a timing file (design_names.del), specific to the layout and timing of your design. The '.del' file contains delay information used by the ALS Timer or other simulators. The Timer is a static analysis tool. With the timer you can get information of all the delays in your design. You can also use it in prelayout mode to get an estimate of the delays before running Place & Route. At the end you can program your device within ALS by downloading the data into the chip.

Simulation of the synthesised design is possible at two points: before and after Place & Route (prelayout and postlayout simulation). This can be done with a lot of third-party software products (Orcad, Susie, VIEWlogic, etc.). Again the most popular tool with ALS design is VIEWlogic's VIEWsim and VIEWwave. Prelayout simulation can be performed from the VL netlist by running the 'vsm' program, which generates the necessary information for VIEWsim. Here unit delays are used for the components. For postlayout simulation the delay information must be back annotated to the simulator (del2adl), and then it will work with real delay information.

5. The Sample Designs

I have used four sample designs to test the synthesis powers of these tools, and to determine the impact of different VHDL constructs. I have translated both behavioural and data-flow descriptions to see if there is any remarkable difference in the output. From now on I will refer to the TI'X'PRESS code as the data-flow construct of CORE, as the two programs will produce the same result for this type of description method. I have tried to use the same source codes for all the software products. This could not be done with every design, because the synthesis tools do not support all VHDL features, and they have their own unique functions and data types. The sample designs are as follows:

- Sixteen bit adder with carry-out;
- Four bit cascadable counter;
- Sixteen bit counter.

5.1 Sixteen Bit Adder with Carry-out

There is no difference between the behavioural and the data-flow description method for combinatorial circuits. I could not use the same code for CORE and Synopsys. I had problems mainly with the carry-out bit, as VHDL cannot handle operations with different sizes, and arithmetic operators are only defined with the integer data type. That is, I could not add two sixteen bit numbers and connect the result to a seventeen bit signal. I have tried to work around the problem, but in the end I was forced to use vendor specific functions. I could have solved this problem by writing my own functions, running with both programs, but it would have been too time consuming. The use of our own functions also causes the tools to generate a lot of unnecessary logic, which is later optimised away, slowing down the translation process.

In CORE I have used the predefined 'add' function, which takes two 'n' bit vectors of std_logic_vector type and returns the result in a '$n+1$' bit vector. Here is the code for the adder:

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.exemplar_1164.all;

entity add16_ti is
  port (a, b  :  in std_logic_vector (15 downto 0);
        c     :  out std_logic_vector (15 downto 0);
        co    :  out std_logic);
end add16_ti;

architecture TI of add16_ti is
  signal s_int :  std_logic_vector (16 downto 0);
begin
  s_int     <= add(a, b);
  co        <= s_int(16);
  s         <= s_int(15 downto 0);
end TI;
```

Synopsys provides a data type ***unsigned***, which is the same as an integer, but its bits can be handled one by one. The software also has arithmetic operators working with this data type. I have used this type for the adder. I also had to use a conversion function (defined in the 'std_logic_arith' package) that extends the larger operand to a seventeen bit signal:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity add16 is
  port (a, b  :  in unsigned (15 downto 0);
        s     :  buffer unsigned (15 downto 0);
        co    :  out std_logic);
end add16;

architecture df_sy of add16 is
signal s_int :  unsigned (16 downto 0);
begin
  process
  begin
  s_int     <= conv_unsigned(a, 17) + b;
  s         <= s_int(15 downto 0);
  co        <= s_int(16);
  end process;
end df_sy;
```

5.2 Sixteen Bit Counter

This is a loadable 16-bit counter with ripple carry output (RCO) and clock enable (CE). First let us see the behavioural description. Here the codes are the same for the two programs:

```
entity cn16 is
  port (Q     :  buffer INTEGER range 0 to 65535;
        P     :  in INTEGER range 0 to 65535;
        RCO   :  out BIT;
        CLK   :  in BIT;
        CLR   :  in BIT;
        CE    :  in BIT;
        LD    :  in BIT );
end cn16;
architecture sy-behav of cn16 is
begin
  process ( CLK, CLR, LD, CE, P, Q )
  begin
    if Q = 65535 then
      RCO <= '1';
    else
      RCO <= '0';
    end if;
    if CLR = '0' then
      Q <= 0;
    elsif (CLK'event and CLK = '1') then
      if LD = '0' then
        Q <= P;
      elsif CE = '1' then
        if Q = 65535 then
          Q <= 0;
        else
          Q <= Q + 1;
        end if;
      end if;
    end if;
  end process;
end;
```

You cannot imply a register with the data-flow description, you can only describe the flow of information between the flip-flops. So the data storage elements must be instantiated or flip-flop functions must be used. With CORE I have used the dffc_ v register function. Here is the input file:

```
entity cn16 is
  port (q     :  out INTEGER range 0 to 65535;
        p     :   in bit_vector (0 to 15);
```

```
         rco    :  out BIT;
         clk    :  in BIT;
         clr    :  in BIT;
         ce     :  in BIT;
         ld     :  in BIT );
end cn16;
architecture ti_df of cn16 is
  signal ff_in, ff_out :  bit_vector(0 to 15);
  signal en :                 bit;
begin
  dffc_v(ff_in, clr, clk, ff_out) ;
  q <= ff_out;
  en        <= (ld and ce);
  rco       <= '1' when q = 65535 else '0';
  ff_in     <= X"0000" when (ld = '1') and (ce = '1')
                      (q = 65535) else
    p            when ld = '0' else
    q+1          when (ld = '1') and (ce ='1') else
    q                 ;
end ti_df;
```

Another way to implement this design is to cascade 4 bit counters to build the 16 bit counter. I have tried this approach with Synopsys, because the synthesis tools usually have problems with more complex logic, so cascading might produce a better result. For cascading I have used a translated and optimised four bit counter described with the behavioural method (you can find the source code in chapter 6). Here is the code that instantiates and routes the smaller counters:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity cn16 is
  port ( clk, clr, ld, ce :  in std_logic;
       p     :  in std_logic_vector (0 to 15);
       q     :  out std_logic_vector (0 to 15);
       rco   :  out std_logic
  );
end cn16;
architecture cascade of cn16 is
  signal rco_0, rco_1, rco_2, one :                     std_logic;
  signal clk_i, clr_i, ce_i, ld_i, rco_i :              std_logic;
  signal p_i, q_i :  std_logic_vector(0 to 15);
  component cn4
    port (CLK, CLR, LD, ENT, ENP : in std_logic;
         P     :  in std_logic_vector (0 to 3);
         Q     :  buffer std_logic_vector (0 to 3);
         RCO   :  out std_logic);
    end component;
    component INBUF
      port (PAD  :  in std_logic;
            Y    :  out std_logic);
    end component;
    component OUTBUF
      port (PAD  :  out std_logic;
            D    :  in std_logic);
    end component;
    component CLKBUF
      port (PAD  :  in std_logic;
            Y    :  out std_logic);
    end component;
  begin
    one <= '1';
    cn4_0:  cn4 port map(clk_i, clr_i, ld_i, ce_i, one,
                         p_i(0 to 3), q_i(0 to 3), rco_0);
    cn4_1:           cn4 port map(clk_i, clr_i, ld_i, one, rco_0,
                         p_i(4 to 7), q_i(4 to 7), rco_1);
    cn4_2:           cn4 port map(clk_i, clr_i, ld_i, rco_1, rco_0,
                         p_i(8 to 11), q_i(8 to 11), rco_2);
    cn4_3:           cn4 port map(clk_i, clr_i, ld_i, rco_2, rco_0,
                         p_i(12 to 15), q_i(12 to 15), rco_i);
    clk_in:       CLKBUF        port map (clk, clk_i);
    clr_in:       INBUF         port map (clr, clr_i);
    ce_in:        INBUF         port map (ce, ce_i);
    ld_in:        INBUF         port map (ld, ld_i);
    rco_in:       OUTBUF        port map (rco, rco_i);
    GEN: for i in q'range generate
    out_q:        OUTBUF        port map (q(i), q_i(i));
    in_p:         INBUF         port map (p(i), p_i(i));
    end generate GEN;
  end;
```

6. The Results of the Analysis

I have optimised the source codes with CORE and Synopsys both for area and speed. I have also tested TI'X'PRESS whether it generates the same output as CORE does, when the input files are the same. I have used the adder for this purpose, and I have found the two results to be identical.

CORE provides two modes of operation controlled by the '-fast' switch. The default mode is optimisation for area. If we set this switch the program will look for the fastest solution. CORE has nine optimisation runs, all of which uses different algorithms, and chooses the best solution according to the design constraints and the mode of operation. CORE also uses different algorithms for speed and area optimisation. It would be possible to reoptimise an ADL netlist more times, and select the best result. I have not done this, because I worked with CORE on a PC, and every pass took approximately 10...30 minutes, depending on the complexity of the design. It was simply too time consuming, so I had to be content with the result of the first optimisation. CORE may run faster on workstations, so reoptimisation could be practical there. The design steps with CORE were the same for all the designs:

- Optimisation:
  Area: fpga design_name.vhd design_name.adl -ta= act2-re=2-adl_check_names
  Speed: fpga design_name.vhd design_name.adl -ta= act2-re=2 -fast -adl_check_names
- ALS Design generation:
  cae2adl -alscknl fam:act2 design_name
  cae2adl -aal2adl aal:d_n.adl design_name

Synopsys is a much more complex system, letting you define a lot of parameters, write libraries, create, view and even edit schematics, simulate VHDL files and so on. You must have the necessary licenses to run the appropriate programs. (The programs I had access to are listed on the front page of the report.) I have used the Design_Analyser for logic synthesis. In this complex system it is not so simple to optimise only for area or speed. I had to play around with the parameters trying to find the fastest or smallest solutions. Here it is worth to rerun the compilation phase over and over again and select the best result, as the optimisation seldom takes more than one minute (it is only true, of course, for these three simple designs). You must set the environment parameters correctly before you start your work. You tell the software which libraries to use, where it should look for files, what conventions it should follow when reading in and writing EDIF, VHDL and other files, and so on. These settings can be done easily by executing a 'script' file. Here is the script file I have used:

```
designer          ="Geza Nemesszeghy" ;
company           ="Texas Instruments" ;
search_path       = {".", "/vol/synopsys/libraries/syn",
"/auto/home/gezan/lib", "/auto/home/gezan/designs/design_lib"}
link_library      = TPC12.db
target_library    = TPC12.db
symbol_library    = TPC12.sdb
edifout_netlist_only = true
edifout_no_array = true
edifout_power_and_ground_representation = cell
edifout_ground_name = GND
edifout_ground_pin_name = Y
edifout_power_name = VCC
```

```
edifout_power_pin_name = Y
edifout_array_member_naming_style = "%s%d"
edifout_array_range_naming_style = "%s_%d_%d"
edifout_match_vhdl_names = "true"
```

It is a problem now in Synopsys to add the input and output buffers to a TPC FPGA design. It is not the fault of the system though. It is possible to specify the different ports as PADs and set the required PAD characteristics, and then the program chooses the appropriate buffer types, or we can explicitly select a buffer type to use. The problem is that the TPC12.db technology library does not include the I/O buffers as ports, so we have to insert these buffers. We can do this in several ways. At first glance it seems that the most straightforward method is to instantiate the PADs in the source code, but this can cause problems. For instantiation we have to be able to access the bits of the signals, so we cannot use integer types, or we must use type conversion functions. I have used a different approach. I have written another VHDL file, where I have instantiated the I/O buffers and the real source code as black boxes. Now I did not have to bother with the data types used in the original design. To make things easier I have used a skeleton file for this purpose, where I only had to add the name and ports of the design. Here are the files of both approaches for the four bit counter:

Instantiating the PADs within the source code:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity cn4_sy is
  port ( CLK_O, CLR_O, LD_O, ENT_O, ENP_O : in std_logic;
         (P_O         :  in std_logic_vector (0 to 3);
         (Q_O         :  out std_logic_vector (0 to 3);
         (RCO_O       :  out std_logic
         );
end cn4_sy;
architecture sy_behav of cn4_sy is
  signal CLK, CLR, LD, ENT, ENP, RCO : std_logic;
  signal P, Q         :  integer range 0 to 15;
  signal P_V, Q_V     :  std_logic_vector (0 to 3);
  component INBUF
    port (PAD  :  in std_logic;
          Y    :  out std_logic);
  end component;
```

```
      component OUTBUF
        port (PAD  :  out std_logic;
             D     :  in std_logic);
      end component;
      component CLKBUF
        port (PAD  :  in std_logic;
             Y     :  out std_logic);
      end component;
    begin
inclk:    CLKBUF   port map   (CLK_O, CLK);
inclr:    INBUF    port map   (CLR_O, CLR);
inld:     INBUF    port map   (LD_O, LD);
inent:    INBUF    port map   (ENT_O, ENT);
inenp:    INBUF    port map   (ENP_O, ENP);
outrco:   OUTBUF   port map   (RCO_O, RCO);
     P_V <= conv_std_logic_vector(P, 4);
     Q_V <= conv_std_logic_vector(Q, 4);
     GEN: for I in P_V'range generate
  inp:     INBUF    port map   (P_O(I), P_V(I));
  outq:    OUTBUF   port map   (Q_O(I), Q_V(I));
     end generate GEN;
     process (CLK, CLR, LD, ENT, ENP, Q)
     begin
       if ((Q = 15) and (ENT = '1')) then
         RCO <= '1';
       else
         RCO <= '0';
       end if;
       if CLR = '0' then
         Q <= 0;
       elsif (CLK'event and CLK = '1') then
         if LD = '0' then
           Q <= P;
         elsif ((ENT = '1') and (ENP = '1')) then
           if Q = 15 then
             Q <= 0;
           else
         Q <= (Q + 1);
           end if;
         end if;
       end if;
     end process;
   end;
```

The source file of the TOP design using the 'skeleton' file (The text added to the 'skeleton file' is written in bold):

```
library IEEE;
use IEEE.std_logic_1164.all;
entity cn4_sy is
  port(q      :  out std_logic_vector (0 to 3);
       rco   :  out std_logic;
       p      :  in std_logic_vector (0 to 3);
       clk, clr, ent, enp, ld:  in std_logic);
end cn4_sy;
architecture synps of cn4_sy is
  signal q_i, p_i:  std_logic_vector (0 to 3);
  signal rco_i, clk_i, clr_i, ent_i,
         enp_i, ld_i:  std_logic;
  component cn4
    port (CLK_O, CLR_O,LD_O, ENT_O, ENP_O : in std_logic;
          P_O      :  in std_logic_vector (0 to 3);
          Q_O      :  buffer std_logic_vector (0 to 3);
          RCO_O :  out std_logic);
  end component;
  component INBUF
    port (PAD  :  in std_logic;
          Y     :  out std_logic);
  end component;
  component OUTBUF
    port (PAD  :  out std_logic;
          D     :  in std_logic);
  end component;
  component CLKBUF
    port (PAD  :  in std_logic;
          Y     :  out std_logic);
  end component;
begin
c4_in:    cn4        port map  (clk_i, clr_i, ld_i, ent_i,
                                enp_i, p_i, q_i, rco_i);
clk_in:   INBUF      port map  (clk, clk_i);
clr_in:   INBUF      port map  (clr, clr_i);
ent_in:   INBUF      port map  (ent, ent_i);
enp_in:   INBUF      port map  (enp, enp_i);
ld_in:    INBUF      port map  (ld, ld_i);
rco_in:   OUTBUF     port map  (rco, rco_i);
  GEN: for i in q'range generate
  out_q:  OUTBUF   port map   (q(i), q_i(i));
  in_p:   INBUF    port map   (p(i), p_i(i));
  end generate GEN;
end;
```

In the following chapters I will first present the results in table format for each design. I will also give the script files and show the steps I have followed in Synopsys, showing the outcome of the different compilation runs, the best one written in bold. The timing and area values given under the Runs heading were generated by Synopsys (the values in the table are from the ALS Timer in postlayout mode). The results of the old versions are from [6]. The version numbers of the tools analysed can be seen below:

Synopsys	→	old version:	V2.2b
		new version:	V3.0
CORE	→	old version:	V1.12
		new version:	V1.21

6.1 Sixteen Bit Adder

Table 1
16 bit adder

Tools		Area		Speed		Old version	
CORE	Sequential modules		0		0		0
	Combinatorial modules		53		118		131
	Longest delay (ns)		172.1		142.4		146.8
	Logic levels		19		16		16
		10.3	OUTBUF	10.3	OUTBUF	12.9	OUTBUF
	Component delay (ns)	5.3	XNOR	5.6	XNOR	8.0	NOR4A
		8.3	XNOR	8.5	AX1C	7.9	XO1
		8.9	MAJ3	8.0	OA5	8.8	MX2A
		9.6	MAJ3	9.3	AOI4A	9.3	AOI4A
		8.9	MAJ3	8.4	AOI4A	9.6	OA5
		9.2	MAJ3	11.2	AOI4A	7.6	OA2A
		10.0	MAJ3	6.9	OA5	7.5	OA5
		7.5	MAJ3	6.3	OA2A	8.9	AOI4A
		9.2	MAJ3	8.4	OA5	11.5	OA5
		9.5	MAJ3	10.1	AOI4A	8.9	OA2A
		75.4	MAJ3	7.5	OA5	6.8	OA5
		9.0	MAJ3	6.7	OA2A	8.5	AOI4A
		57.5	MAJ3	9.3	OA5	8.9	OA5
		10.9	MAJ3	8.6	AOI1	8.7	AND2
		9.2	MAJ3	17.3	INBUF	14.4	INBUF
		6.9	AO1C				
		6.5	NAND3				
		14.8	INBUF				

Optimisation for area: First you must read in the VHDL file and set the parameters by running the script file given below. Then you should compile the design three times without changing the settings.

Script file:

```
current_design = add16
set_flatten true ;
set_flatten -effort medium ;
set_flatten -minimize single_output ;
set_flatten -phase false ;
set_structure true -boolean true;
set_local_link_library {TPC12.db} ;
max_area 0 ;
```

Table 2
16 bit adder

Tools		Area		Speed		Old version	
SYNOPSYS	Sequential modules	0		0		0	
	Combinatorial modules	46		110		170	
	Longest delay (ns)	163.7		83.2		112.7	
	Logic levels	17		9		11	
		10.3	OUTBUF	10.3	OUTBUF	8.9	OUTBUF
	Component delay (ns)	5.6	XNOR	7.3	AX1B	7.2	AND5B
		8.6	MAJ3	7.8	AO6A	11.6	AND4
		13.3	MAJ3	8.6	INV	9.7	XNOR
		8.4	MAJ3	7.3	INV	6.7	AOI1B
		9.2	MAJ3	8.9	MAJ3	9.3	OA2A
		8.9	MAJ3	8.9	AO7	8.2	AND2A
		10.0	MAJ3	7.6	AND2	6.6	OA3A
		10.1	MAJ3	16.5	INBUF	8.3	OR2B
		9.8	MAJ3			6.8	OR2B
		9.0	MAJ3			29.4	INBUF
		8.9	MAJ3				
		8.5	MAJ3				
		9.7	MAJ3				
		10.7	MAJ3				
		10.0	CY2A				
		12.7	INBUF				

Runs: (compile -map_effort high)

AREA (cells)	TIME (ns)
55	292.30
49	223.29
46	**188.79**
51	245.98
53	267.70

– *Optimisation for speed:* After reading in the VHDL file and setting the parameters you only have to run the optimisation phase once.

Script file:

```
current_design = add16
max_delay 0 all_outputs
set_flatten false ;
set_structure true ;
set_local_link_library {TPC12.db};
set_ungroup find(cell, "plus");
```

Runs: (compile -map_effort high)

TIME (ns)	AREA (cells)
79.87	**110**
81.63	125

6.2 Sixteen Bit Counter

– *Optimisation for area:* Synopsys has more problems with sequential logic. It is more difficult to reach an acceptable result, and you can never be sure that you have the best output that the software can produce. I could not reach a good result for area optimisation. After setting the parameters you should compile 4 times, then set the timing driven structuring off and compile again.

Script file:

```
current_design = cn16
set_flatten true ;
set_flatten -effort medium ;
set_flatten -minimize multiple_output ;
set_flatten -phase true ;
```

Table 3
16 Bit counter

Tools		Cascading		Behavioral		
		area	speed	area	speed	old
SYNOPSYS	Sequential modules	16	16	16	16	16
	Combinatorial modules	35	54	60	112	92
	Max. frequency (MHz)	12.22	22.88	7.22	21.5	19.76
	Logic levels	11	5	18	6	6
		0.5 DFMB	0.5 DFMB	0.5 DFMB	0.5 DFC1B	0.5 DFMB
Component delay (ns)		7.4 XOR	7.8 XOR	7.8 TA157	0.0 AO5A	6.1 OA2
		7.3 NAND2	7.7AND4A	6.9 AND2A	6.6 NAND4A	8.6 AND2B
		9.0 HA1	14.1 OR4D	6.5 AO1	10.4 AO6A	10 NOR4D
		8.4 AND3A	13.2DFMB	10.0 AND2A	9.3 NOR3C	11.3 NOR4D
		7.4 NAND2	0.4 skew	7.9 NAND2	13.6 DFC1B	13.3 DFMB
		8.4 AND2A		9.5 AND2	6.0 skew	
		8.9 NAND3A		10.3 AND2		
		7.1 NAND2		9.3 AND2		
		7.9 HA1		9.0 AX1B		
		9.1 DFMB		9.0 AND2A		
		0.4 skew		6.6 AX1B		
				6.5 INV		
				8.0 AX1B		
				6.3 INV		
				7.8 AX1C		
				7.2 INV		
				8.7 DFMB		
				0.7 skew		

Table 4
16 Bit counter

Tools		Data-flow		Behavioral		
		area	speed	area	speed	old
CORE	Sequential modules	16	16	16	16	16
	Combinatorial modules	40	44	40	45	61
	Max. frequency (MHz)	13.3	24.87	13.6	19.61	19
	Logic levels	9	5	9	6	6
		0.5 DFMB	0.5 DFMB	0.5 DFM	0.5 DFMB	0.5 DFMB
Component delay		7.5 AX1C	6.9 AX1C	7.4 AX1	7.8 AX1C	6.5 AX1C
(ns)		7.9 AND2	10.4 AND5B	8.1 AND	8.4 AND5B	9.5 AND4
		8.4 AND2	8.4 INV	8.8 AND	9.5 AND5B	10.8 DFMB
		8.5 AND2	12.6 DFMB	8.0 AND	11.0 AND4	11.3 AND4
		9.1 AND2	1.4 skew	8.4 AND	12.7 DFMB	11.6 DFMB
		11.3 AND2		11.9 AND2	1.1 skew	
		10.8 AND4		10.7 AND4		
		12.5 DFMB		11.8 DFMB		
		-1.4 skew		-2.2 Psk		

Table 5
4 Bit counter

Tools		Behavioral		old
		area	speed	
SYNOPSYS	Sequential modules	4	4	4
	Combinatorial modules	10	10	27
	Max. frequency (MHz)	38.17	38.17	17.89
	Logic levels	4	4	8
	Component delay (ns)	0.5 DFMB 7.1 XOR 7.9 AND4 10.7 DFMB 0.0 skew	0.5 DFMB 7.1 XOR 7.9 AND4 10.7 DFMB 0.0 skew	0.5 DFE3A 6.2 INV 8.5 OA2A 7.3 XOR 9.3 NAND2 7.2 INV 9.3 INV 7.6 DFC1B

```
set_structure true -boolean true -timing -true;
set_register_type -exact -flip_flop DFMB ;
max_area 0 ;
set_ungroup find( cell, "adder") true ;
```

Runs: (compile -map_effort high)

AREA (cells)	TIME (ns)
87	363.22
81	208.26
80	253.63
77	229.03
set_structure true -Boolean true -timing false	
76	**202.86**

– *Optimisation for speed:* The result for speed optimisation is quite good when we compare it to the design with the minimum area. Here you only have to rerun the compilation phase three times to achieve this result.

Script file:

```
cn16_sys.db:cn16_sys
current_design = cn16
create_clock -period 50 _waveform {0 25} find(port,"CLK")
```

```
max_delay 0 all_registers(-data_pins) + all_outputs()
set_flatten true ;
set_flatten -effort medium ;
set_flatten -minimize single_output ;
set_flatten -phase false ;
set_structure true ;
set_register_type -flip_flop DFM6B ;
set_max_fanout 24 all_inputs() ;
set_ungroup find( cell, "adder") true ;
```

Runs: (compile -map_effort high)

ARE (cells)	**TIME** (ns)
138	76.66
142	64.84
128	**64.84**

– *Cascading the 4 bit counters:* I have also tried cascading to build the 16 bit counter. I translated the 4 bit counter first, where I used the FSM (Finite State Machine) extraction. After reading in the files I reoptimised the design both for area and speed. I have grouped the four smaller counters into one design (see the script files) excluding the PADs. I had to do this to prevent Synopsys to connect a net to the wrong port of the I/O buffers. Here are the script files I have used:

Script file for area optimisation:

```
current_design = cn16.db:cn16
create_clock -period 50 -waveform {0 25} find(port,"clk")
set_ungroup find( cell, "cn4_0") true ;
set_ungroup find( cell, "cn4_1") true ;
set_ungroup find( cell, "cn4_2") true ;
set_ungroup find( cell, "cn4_3") true ;
group {"cn4_0", "cn4_1", "cn4_2", "cn4_3"} _design_name
cn16_sy_cas_core ;
current_design = cn16_sy_cas_core
max_area 0 ;
```

Runs: (compile -map_effort high)
medskip

AREA (cells)	TIME (ns)
51	(not recorded)
50	**121.07**
50	131.94

Script file for speed optimization:

```
current_design = cn16
set_ungroup find( cell, "cn4_0") true ;
set_ungroup find( cell, "cn4_1") true ;
set_ungroup find( cell, "cn4_2") true ;
set_ungroup find( cell, "cn4_3") true ;
group {"cn4_0", "cn4_1", "cn4_2", "cn4_3"} _design_name
cn16_sy_cas_core ;
current_design = cn16_sy_cas_core
create_clock -period 50 -waveform { 0 25 }
find(port,"clk_i")
```

Runs: (compile -map_effort high)

AREA (cells)	TIME (ns)	*(compile cn16_sy_cas_core only)*
61	57.19	
65	**55.74**	
63	61.02	
62	60.06	
65	60.06	

7. Conclusion

We can derive a lot of useful information from the results presented above. They can be helpful when choosing from the analysed synthesis tools, and they can also show the areas where the these tools need to be improved.

There is considerable difference between Synopsys and CORE. Synopsys is a highly complex system created mainly for ASIC design. This complexity makes it more flexible, but also more difficult to use. If you want to get good results out of this system you must spend hours of hard work with it. When you are using Design Analyser you can try a lot of ways to realise your design. The only problem is that if you do not save and document every single compilation phase you will find it really difficult to reproduce a result previously attained, especially with large designs. We must also be aware of this when we compare the old and new versions. The better results of the new release may be only due to the considerable amount of time that I have spent in improving the designs.

CORE was written mainly for FPGAs. The advantage of this VHDL compiler lies in its simplicity. It can run on PCs as well, so you only need a single Personal Computer to design FPGAs. You can learn all you have to know about this software within one day. This program has two drawbacks:

it does not allow as much control of the synthesis as Synopsys does, and the translation takes a lot of time, at least on PCs. We have to take into account the fact that the area and speed values for CORE are the results of only one compilation run, while I made a lot of translations and trials with Synopsys.

It is quite evident from the results that Synopsys is smarter with *arithmetical designs*. It could produce better gate-utilisation and speed values even for the first compilation. We can observe an improvement on the old versions, especially with Synopsys. CORE seems to work better with sequential designs, as it gave better results for the *counters*. I had to try a lot of settings in Design Analyser to attain the result I have presented. It is interesting that the FSM (Finite State Machine) extraction yielded the best values for the 4 bit counter both for area and speed. This means that Synopsys can translate a state table format more efficiently than a VHDL file. Unfortunately the extraction did not work for the 16 bit counter, as the system always broke down when I tried to extract the state machine. Probably it cannot cope with a 16 bit long state vector.

We would expect that the *data-flow* description, being less abstract, should produce better results than the behavioural, and we can see a considerable difference between the results at the speed optimisation of the relatively complex 16 bit counter. So we might get better area and speed values for larger designs if we use the data-flow description.

Cascading shows that it is worth mixing the different VHDL description methods. I have reached the best results here with Synopsys both for area and speed optimisation. I could synthesise the 16 bit counter using the minimum number of logic cells (51), which is even less than the values achieved with CORE. Cascading may also be tried with CORE using the EIL file format.

7. HDL Textbooks

IEEE Standard VHDL Language Reference Manual. IEEE Std 1076–1987., Published by The Institute of Electrical and Electronics Engineers, Inc. 345 East 47th Street, New York, NY 10017, U.S.A.
VHDL: Hardware Description and Design, Lipsett, Schaefer, Ussery, Kulwer Academic Publishers, 101 Philip Drive, Assinipi Park, Norwell, Massachusetts 02061 U.S.A.
The VHDL handbook, David R. Coelho, Kulwer Academic Publishers, 101 Philip Drive, Assinipi Park, Norwell, Massachusetts 02061 U.S.A.
Chip Level Modeling with VHDL, James R. Armstrong, Prentice Hall, Englewood Cliffs, New Jersey 07632 U.S.A

VHDL, Douglas L. Perry, McGraw-Hill, Inc. Professional Publishing Group, 11 West 19th Street, New York, NY 10011 U.S.A
Application of VHDL to CErcuit Design, Harr and Stanculescu, Kulwer Academic Publishers, 101 Philip Drive, Assinipi Park, Norwell, Massachusetts 02061 U.S.A.
Introduction to HDL — based design using VHDL, Steve Carlson, Synopsys, Inc. 1098 Alta Avenue, Mountain View California 94043
The Verilog Hardware Description Language, Donald E. Thomas, Philip Moorby, Kulwer Academic Publishers, 101 Philip Drive, Assinipi Park, Norwell, Massachusetts 02061 U.S.A.
Digital Design with Verilog HDL, Eliezer Sternheim, Rajvir Singh, Yatin Trivedi, Design Automation Series 1990, Automata Publishing Company, Cupertino, CA 95014 U.S.A.

References

1. Synopsys VHDL Compiler Reference Manual (Version 3.0 1992, Synopsys, Inc.)
2. Synopsys DesignWare Databook (Version 3.0 1992, Synopsys, Inc.)
3. Digital Design with Verilog HDL (Eliezer Sternheim, Rajvir Singh, Yatin Trivedi, Design Automation Series 1990, Automata Publishing Company, Cupertino, CA 95014 U.S.A.)
4. Mentor Graphics Introduction to VHDL (Mentor Graphics Corporation 8005 S.W. Boeckman Road, Willsonville, Oregon 97070, December 1991)
5. Exemplar VHDL Synthesis Reference Manual (Exemplar Logic, Inc. 2250 Ninth Street, Suite 102 Berkeley, CA 94710, 1993)
6. How well can Synthesis Tools map Logic to FPGAs (Dr. Dung Tu, Technical Marketing Manager, Texas Instruments)