A Primer to Digital Design with Synopsys and Cadence

Martin Horauer

Salvador Dali, "Die Beständigkeit der Erinnerung"
Contents

1 Synopsys .................................................. 2
   1.1 Synopsys documentation .................................. 2
   1.2 Analyzing source files .................................. 2
   1.3 Directory hierarchy and file-naming policy ............... 3
   1.4 Simulation before synthesis .............................. 4
   1.5 Synthesis ............................................... 5
   1.6 A design example ......................................... 7

2 Back-End design with CADENCE ............................. 25
Chapter 1

Synopsys

Here we will give a brief description of the basic programs and shells, that can be used in conjunction with an ASIC development. I will not cover the use of the schematic entry tool SGE, that is (that's just another prejudice of mine) preferably useful for interconnect management at the top level.

![Figure 1.1: SGE fronted after invocation](image)

If you're intending to make use of this tool, see the online documentation for an extensive description.

1.1 Synopsys documentation

The large collection of the Synopsys manuals can be read and searched online via the very powerful inter-leaf viewer. Type `view` at the Unix command prompt to bring it up onto your screen. Figure 1.2 shows the outline after invocation. Very powerful context search is available via `Search -> Collection`. This documentation is an excellent guide to further work, so I’ll recommend it for your use not only when you're a novice. Furthermore there is a collection of books available at our department, that are primarily related to HDL coding [R.94], [P.90], [J.92], [HYE95], [R.93], [D.91], [IEE87], [IEE93], [PT95] and [S.96].

1.2 Analyzing source files

The program `vhdlan` can be used to analyze the VHDL-files for correct syntax.

```bash
% vhdlan -nc vhdl/test.ent.vhd vhdl/test.beh.arch.vhd
```
The `-nc` option avoids just an annoying header displayed at every invocation. Beside vhdlan has many other useful options and switches. See the online help `iview` or type `vhdlan --help` to get more information on them. When you're dealing with several source files that are depending on each other you can generate a `Makefile` using the command `simdepends`. Once you've got your `Makefile`, you'll only need to enter `make`, and all depending VHDL-source files, that were modified more recently, are analyzed again. In the case when the analysis of your code succeeds, you can start with your simulation. Otherwise the lines and locations of the error prone code fragments are displayed. You'll have to identify the error an rerun the analysis.

### 1.3 Directory hierarchy and file-naming policy

To ease file manipulation and navigation through a project, everybody in the design team should follow some general naming and structure conventions. The following directory tree serves as an example how to structure the design.

```
/project/dd
   syn
   mdu tdu edu ... scr comp scan s2c ver
   cds
       top_dd.run ...
       wrfDir
       shmDir
```

The `cds` directory holds all Cadence related files, while `syn` all Synopsys related. In the
Synopsys tree the subdirectories mdu, tdu and exu hold the VHDL-source files for the three modules of the design. The directory scr contains all related script files for synthesis and simulation, under comp all synthesized files are stored and scan holds all files after scan-path insertion. Finally the contents of directory s2c are made up with several files necessary for the design transfer to Cadence (physical place and route).

In the Cadence tree, many subdirectories are instantiated by the tools and therefore you should avoid to manipulate to much in there. Use the tools supplied with Cadence instead.

<table>
<thead>
<tr>
<th>filename</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dd.pkg.vhd</td>
<td>project package declaration</td>
</tr>
<tr>
<td>dd.pbdy.vhd</td>
<td>project package body</td>
</tr>
<tr>
<td>exu.ent.vhd</td>
<td>top-module entity</td>
</tr>
<tr>
<td>exu.beh_arch.vhd</td>
<td>top-module architecture</td>
</tr>
<tr>
<td>exu.beh_cfg.vhd</td>
<td>top-module configuration</td>
</tr>
<tr>
<td>exu.body_ent.vhd</td>
<td>sub-module entity (body varies)</td>
</tr>
<tr>
<td>exu.body_beh_arch.vhd</td>
<td>sub-module architecture</td>
</tr>
</tbody>
</table>

Table 1.1: Filenaming policy

Table 1.1 summarizes a possible filenaming convention. It is of course a bit tedious to split everything, but benefits of a highly modular design should always be kept in mind. Beside, every file should contain a header, that includes the project name, the name of the designer, the version number, the filename itself, the title and type of the module, the tools it is targeted for, which libraries and packages are used and of course a timestamp. A few of these header lines are of course obvious and could be taken from the file properties itself, but they are nice to have when included also in the header. For examples consult the source-code listings in the further sections.

### 1.4 Simulation before synthesis

Within Synopsys either the graphical interface vhdlbx (see Figure 1.3) or the command line version vhdsim can be used. In both cases you need to specify the name of the configuration you’re intending to simulate. Afterwards you’ll have to specify the signals you’re willing to trace. The later can also be specified in an include file that is applied to the simulator during invocation. You can single-step the code, set/delete breakpoints, evaluate signals and variables, set them to specific values, interrupt the simulation, …. In the waveform window you can add several cursors to measure timings between different signals, zoom in and out, …. Furthermore you can implement file i/o, to ease comparison of your simulation results at different levels of simulation. The later can in addition also be performed via a post-processor program gpp. Gpp takes two wif files (file-extension .ow), and compares them at periodic times you specify at the command line. A wif file is automatically created when the variable WAVEFORM is set to wif+waves within the .synopsys.vss_setup configuration file.

The simulator is usually invoked with an include file, which consists of several control sequences for the simulator itself. The following gives an idea of such a file. It traces several signals in different levels of the hierarchy and displays them in the waveform window.
An analogue file for the post-processor could be for e.g. as the following.

Here two wif files, that must be created previously with the simulator, are loaded first. The files command displays all files in memory. The timespec command declares a periodic point in simulation time, starting at time 40530 and repeating 40000 times every 500 time increments. The resolution for the time increments (ns, ps, ...) is determined via the setup-file .synopsys.vss_setup.

1.5 Synthesis

The programs for synthesis are the graphical user interface design_analyzer (see Figure 1.4) and the command line shell dc.shell. Both have the same functionality, although the shell version is of course more appropriate for most synthesis runs, while the graphical interface can be used primarily for cross-checkings and is easier to use by new users. Both programs have many options and commands built-in. A few of them will be illustrated a little along with some scripts we'll illustrate along with the following sections.
To get more information on specific commands just type `help <topic>` at the shell prompt or within the command window.

Figure 1.4: The graphical user interface for synthesis

To illustrate the functionality a bit more in depth, take a look at the following example.

```
PACKAGES=(plglUnix.pkg.vhd plglUnix plut.vhd)
analyze if vhd in PACKAGES
/* BTU */
BTU.VHDLFILES=(bluethumbadder.bin vhd blueuaddadder unbe.bin.sdr vhd)
bblueu.body vhd bluethumbaddadder unbe.bin.sdr vhd
bblueu.mdnxv.vhd bblueu.mdnxv unbe.bin.sdr vhd
bblueu.u36v.vhd bblueu.u36v unbe.bin.sdr vhd
analyze if vhd in BTU.VHDLFILES
elaborate BTU
...current designs(NIMADDER)
setmaxdelay(28 n,outputs)
compile -ungroup, all
```

First a `dc_shell` variable `PACKAGES` is defined that consists of two files. They are analyzed in the subsequent line. Within this package you could have defined several constants or functions that are used from within several other source files. The subsequent variable holds all VHDL-files of the module in a bottom up fashion. These files are analyzed and elaborated. In the case this operation succeeds, the module is ready to accept several constraints for logic synthesis. Afterwards a submodule `NTPADDER`, that consists of pure combinational logic is synthesized in front of the rest. Therefore you’ve set the current design to this submodule. Then the outputs of this submodule are constrained with a maximum output delay of 28 ns. -- Use a realistic value for constraining, otherwise you'll end up with an unnecessarily huge bulk of logic and in addition you'll waste much processing power of your workstation. -- The submodule then is synthesized and ungrouped, so that you'll end up with a flat logic.

If the module has registered inputs and outputs the following sequence would be an appropriate constraining scenario.
After the current design is set, a variable `clock.name` is defined with the name of the clock signal. In this case the name of the clock signal is `ITCLK`. Then a clock constraint is instantiated, that defines the clock with a period of 40 ns and the rising edge set to 0 ns and the falling to 20 ns. The next constraint `set_dont_touch_network` prevents the instantiation of buffers within this network. – For example, `ES2` uses only one large buffer, strong enough to drive the whole clock tree. Such a buffer is usually instantiated manually after synthesis. – Then according to the technology library you’re syntesizing for, you’ll have to set `operating_conditions` and provide a `wire_load_model` for area estimation of the interconnections. The next two commands are used if you’re intending to insert scan-path logic after synthesis. They set some restrictions for the optimisation of sequential cells to ease the scan-path insertion afterwards. – In the case, you’re willing to omit scan insertion, leave these commands aside. – Then set input and output delay constraints in relation to the clock edge to all appropriate signals. – The `set_input_delay` defines the delay of the path to an input. The value is the total time a signal takes to propagate through logic in front of the input port. In contrast the `set_output_delay` value is the delay through the logic hooked to the output port. The common time reference is the rising edge of the clock signal. Include library setup time and instance-specific clock skew in this calculation. – Use the `set_load` command with library dependent typical values to constrain the output ports, and the `set_max_fanout` for the input ports. An inverter is than specified as a `driving_cell` for the corresponding inputs. Again these attributes are removed from the clock network, because it should be considered special. Finally compile the module. The switch `incremental_map` takes care of the already precompiled submodule. With the `write` command you are able to save the results in different formats, e.g. vhdl, verilog or the Synopsys internal database format (.db). After synthesis you usually create several reports to verify if the synthesized results meet your desired requirements.

If you’re unaware of the meaning or usage of certain commands, you can type `help <command>` at the `dc.shell` prompt or contact the online documentation `iview` from the Unix prompt.

### 1.6 A design example

To illustrate the design flow and to make it a bit more clear and concise imagine the following example displayed in figure 1.5. This example consists of two submodules, the `EXU.BODY`, that holds the core functionality of the module, and a submodule called `EXU.MOMUX`, that is split from the body, due to the fact that it is for e.g. reused within other modules. The block diagram entails the interconnect lines to other modules, and
Figure 1.5: A design example: Example Unit (EXU)

gives a rough impression on the internal structure of the module.

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

entity exu is
port(
    ADDR: in std_logic_vector(1 downto 0);
    DADDR: in std_logic;
    REG: in std_logic_vector(3 downto 0);
    CLK: in std_logic;
    REN: in std_logic;
    READ1: in std_logic_vector(1 downto 0);
    WRITE1: in std_logic_vector(1 downto 0);
    READ2: in std_logic_vector(3 downto 0);
    WRITE2: in std_logic_vector(3 downto 0);
    THROUGH: in std_logic;
    TEAMMODE: in std_logic;
    ENTR: in std_logic;
    ENTR1: out std_logic;
    ENTR2: out std_logic;
    UNITY: out std_logic;
);
end exu;
```

This file which holds the entity of the EXU top module, is saved in the directory `exu` under the filename `exu_ent.vhd`. 
The architecture of the EXU top module is saved as exu/exu_beh_arch.vhd. Finally the configuration is stored as exu/exu_beh_cfg.vhd.
Figure 1.6: Schematic of the submodule EXU_MDMUX

Now that the top elements are specified, let's work out the interior of the submodules. Figure 1.6 entails the schematic of the module EXU_MDMUX, which consists of only one large multiplexer, that feeds one of four 32-bit wide buses, depending on the value of the lines READMUX to the output MDATAR. For larger modules a table should summarize the functionality, such as provided with the following submodule, to ease understanding and to write down a functional mapping. The following code saved in exu/exu.mdmux_ent.vhd holds the entity.

And in addition exu/exu.mdmux_beh_arch.vhd specifies the architecture.
The second submodule EXU.BODY consists of an input bus EXUSB, that provides the
EXU with varying data. The registers MSRCV and TSRCV sample the data of this bus on
the activation of the gate signal TSRCVGATE, that is externally activated synchronous
to the bus. In analogy, the registers MSXMT and TSXMT are sampled by TSXMTGATE.
Both signals are additionally fed through this unit (INTTSRCV and INTTSXMT) to a fol-
low up module. All these four 32-bit registers can than be read via a 32-bit access from the
address MDATA[2X].

At the bottom, there are two registers DUTH and DUTYL located, that can be written.
The 32-bit DUTH and the lower 16 bits of DUTYL serve as DUTY input for a compar-
ator module. Bit 17 of register DUTH enables or disables this comparator, that performs a
comparison of 48 bits between EXUSB[55,8] and the Duty value. When the Exus
is greater than or equal to Duty, a pulse is generated on the output-line INTDUTY, that
is active for one clock period, whilst signal SDUTY is active as long as the EXUSB ≥
DUTY.

<table>
<thead>
<tr>
<th>Element</th>
<th>Width</th>
<th>R/W</th>
<th>ADDR</th>
<th>BE</th>
<th>Reset</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSRCV</td>
<td>32</td>
<td>R</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>Macrostamp receive</td>
</tr>
<tr>
<td>TSRCV</td>
<td>32</td>
<td>R</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>Timestamp receive</td>
</tr>
<tr>
<td>MSXMT</td>
<td>32</td>
<td>R</td>
<td>2</td>
<td>x</td>
<td>0</td>
<td>Macrostamp transmit</td>
</tr>
<tr>
<td>TSXMT</td>
<td>32</td>
<td>R</td>
<td>3</td>
<td>x</td>
<td>0</td>
<td>Timestamp transmit</td>
</tr>
<tr>
<td>DUTH</td>
<td>32</td>
<td>W</td>
<td>0</td>
<td>3:0</td>
<td>0</td>
<td>DUTHY high</td>
</tr>
<tr>
<td>DUTYL</td>
<td>17</td>
<td>W</td>
<td>1</td>
<td>2:0</td>
<td>0</td>
<td>Enable and DUTYL</td>
</tr>
</tbody>
</table>

The following listings belong to the files exu/exu.body_ent.vhd and
exu.body_beh.arch.vhd.
Figure 1.7: Schematic of the submodule EXU_BODY
ARCHITECTURE Behavioral of exu_body IS

  SIGNAL DOUTY : std_logic_vector(3 downto 1);
  SIGNAL DOUTX : std_logic_vector(5 downto 1);
  SIGNAL ES : std_logic;
  SIGNAL LSOUTY : std_logic;
  SIGNAL TRENDAUTO, TRENDBOT : std_logic;

  BEGIN

    -- Initialization

    PROCESS P_INIT
    BEGIN
      LSOUTY := '1';
      TRENDAUTO := '1';
      TRENDBOT := '0';
      DOUTY := (others => '0');
      DOUTX := (others => '0');
      ES := '0';
      REPORT "Initialization complete" SEVERITY "NOTE";
    END PROCESS P_INIT;

    -- Process P_EXBG

    PROCESS P_EXBG
    BEGIN
      IF (RESET='1') THEN
        ES := '0';
        LSOUTY := '0';
        TRENDAUTO := '0';
        TRENDBOT := '0';
        DOUTY := (others => '0');
        DOUTX := (others => '0');
      ELSE
        IF (CLEAR='1' AND CLK='event') THEN
          IF (TRENDAUTO='1') THEN
            ES := '1';
            LSOUTY := '0';
            TRENDAUTO := '0';
            TRENDBOT := '0';
            DOUTY := (others => '0');
            DOUTX := (others => '0');
          END IF;
        END IF;
      END IF;
    END PROCESS P_EXBG;

    -- Process P_TRENDAUTO

    PROCESS P_TRENDAUTO
    BEGIN
      IF (CLEAR='1') THEN
        DOUTY := (others => '0');
        DOUTX := (others => '0');
        LSOUTY := '0';
        TRENDAUTO := '1';
        TRENDBOT := '0';
      ELSE
        IF (CLEAR='1' AND CLK='event') THEN
          IF (TRENDAUTO='1') THEN
            DOUTY := (others => '0');
            DOUTX := (others => '0');
            LSOUTY := '1';
            TRENDAUTO := '0';
            TRENDBOT := '0';
          END IF;
        END IF;
      END IF;
    END PROCESS P_TRENDAUTO;

    -- Process P_TRENDBOT

    PROCESS P_TRENDBOT
    BEGIN
      IF (CLEAR='1') THEN
        DOUTY := (others => '0');
        DOUTX := (others => '0');
        TRENDAUTO := '0';
        TRENDBOT := '1';
        LSOUTY := '0';
      ELSE
        IF (CLEAR='1' AND CLK='event') THEN
          IF (TRENDBOT='1') THEN
            DOUTY := (others => '0');
            DOUTX := (others => '0');
            TRENDAUTO := '0';
            TRENDBOT := '0';
          END IF;
        END IF;
      END IF;
    END PROCESS P_TRENDBOT;

    -- Process P_DOUTY

    PROCESS P_DOUTY
    BEGIN
      IF (CLEAR='1') THEN
        DOUTY := (others => '0');
      ELSE
        IF (CLEAR='1' AND CLK='event') THEN
          IF (TRENDAUTO='1' OR TRENDBOT='1') THEN
            DOUTY := (others => '0');
          ELSE
            DOUTY := (others => '0');
          END IF;
        END IF;
      END IF;
    END PROCESS P_DOUTY;

    -- Process P_DOUTX

    PROCESS P_DOUTX
    BEGIN
      IF (CLEAR='1') THEN
        DOUTX := (others => '0');
      ELSE
        IF (CLEAR='1' AND CLK='event') THEN
          IF (TRENDAUTO='1' OR TRENDBOT='1') THEN
            DOUTX := (others => '0');
          ELSE
            DOUTX := (others => '0');
          END IF;
        END IF;
      END IF;
    END PROCESS P_DOUTX;

    -- Process P_ES

    PROCESS P_ES
    BEGIN
      IF (CLEAR='1') THEN
        ES := '0';
      ELSE
        IF (CLEAR='1' AND CLK='event') THEN
          IF (TRENDAUTO='1' OR TRENDBOT='1') THEN
            ES := '1';
          ELSE
            ES := '0';
          END IF;
        END IF;
      END IF;
    END PROCESS P_ES;

    -- Process P_TRENDAUTO

    PROCESS P_TRENDAUTO
    BEGIN
      IF (CLEAR='1') THEN
        DOUTY := (others => '0');
        DOUTX := (others => '0');
        LSOUTY := '0';
        TRENDAUTO := '1';
        TRENDBOT := '0';
      ELSE
        IF (CLEAR='1' AND CLK='event') THEN
          IF (TRENDAUTO='1') THEN
            DOUTY := (others => '0');
            DOUTX := (others => '0');
            LSOUTY := '1';
            TRENDAUTO := '0';
            TRENDBOT := '0';
          END IF;
        END IF;
      END IF;
    END PROCESS P_TRENDAUTO;

    -- Process P_TRENDBOT

    PROCESS P_TRENDBOT
    BEGIN
      IF (CLEAR='1') THEN
        DOUTY := (others => '0');
        DOUTX := (others => '0');
        TRENDAUTO := '0';
        TRENDBOT := '1';
        LSOUTY := '0';
      ELSE
        IF (CLEAR='1' AND CLK='event') THEN
          IF (TRENDBOT='1') THEN
            DOUTY := (others => '0');
            DOUTX := (others => '0');
            TRENDAUTO := '0';
            TRENDBOT := '0';
          END IF;
        END IF;
      END IF;
    END PROCESS P_TRENDBOT;

    -- Process P_DOUTY

    PROCESS P_DOUTY
    BEGIN
      IF (CLEAR='1') THEN
        DOUTY := (others => '0');
      ELSE
        IF (CLEAR='1' AND CLK='event') THEN
          IF (TRENDAUTO='1' OR TRENDBOT='1') THEN
            DOUTY := (others => '0');
          ELSE
            DOUTY := (others => '0');
          END IF;
        END IF;
      END IF;
    END PROCESS P_DOUTY;

    -- Process P_DOUTX

    PROCESS P_DOUTX
    BEGIN
      IF (CLEAR='1') THEN
        DOUTX := (others => '0');
      ELSE
        IF (CLEAR='1' AND CLK='event') THEN
          IF (TRENDAUTO='1' OR TRENDBOT='1') THEN
            DOUTX := (others => '0');
          ELSE
            DOUTX := (others => '0');
          END IF;
        END IF;
      END IF;
    END PROCESS P_DOUTX;

    -- Process P_ES

    PROCESS P_ES
    BEGIN
      IF (CLEAR='1') THEN
        ES := '0';
      ELSE
        IF (CLEAR='1' AND CLK='event') THEN
          IF (TRENDAUTO='1' OR TRENDBOT='1') THEN
            ES := '1';
          ELSE
            ES := '0';
          END IF;
        END IF;
      END IF;
    END PROCESS P_ES;
  END ARCHITECTURE;
IF [TERM_CODE] = '1' THEN
    TRMP := EXCUSE[1] downto 0;
END IF;
END IF;
END PROCESS P_TTERM;

---

PROCESS: P_SUTY
----------
Purpose:
Inputs: ADDR, SC, ADDST.CLE, MANTYAW
Outputs: DUTFM

P_SUTY : PROCESS (ADDR, SC, ADDST.CLE, MANTYAW)
BEGIN
    IF (ADDST=0'1) THEN
        DUTFM <=古い;
    ELSE
        IF (CLE='1' and CLA_event) THEN
            IF (ADDR='1' and MANTYAW='0' and SC='0') THEN
                DUTFM(15 downto 0) <= MANTYAW(15 downto 0);
            ELSIF (ADDR='1' and MANTYAW='0' and SC='1') THEN
                DUTFM(15 downto 0) <= MANTYAW(15 downto 0);
            ELSE
                DUTFM(15 downto 0) <= MANTYAW(15 downto 0);
            END IF;
        END IF;
    END IF;
END PROCESS P_SUTY;

---

PROCESS: P_DUTY
----------
Purpose:
Inputs: ADDR, BA, ADST, CLE, MANTYAW
Outputs: DUTFYE

P_DUTY : PROCESS (ADDR, BA, ADST, CLE, MANTYAW)
BEGIN
    IF (ADDST=0'1) THEN
        DUTFYE <=古い;
    ELSE
        IF (CLA='1' and CLE_event) THEN
            IF (ADDR='1' and MANTYAW='0' and SC='0') THEN
                DUTFYE(15 downto 0) <= MANTYAW(15 downto 0);
            ELSIF (ADDR='1' and MANTYAW='0' and SC='1') THEN
                DUTFYE(15 downto 0) <= MANTYAW(15 downto 0);
            ELSE
                DUTFYE(15 downto 0) <= MANTYAW(15 downto 0);
            END IF;
        END IF;
    END IF;
END PROCESS P_DUTY;

---

PROCESS: P_DUPY
----------
Purpose:
Inputs: DUTFYE, DUTFML, EXCUSE.CLE, CLE, ADST
Outputs: IMPUTY, DUTFY

P_DUPY : PROCESS (DUTFYE, DUTFML, EXCUSE.CLE, CLE, ADST)
BEGIN
    IMPUTY <= [IMPUTY and [CLE and DUTFML]]
    DUTFY <= TRMPTY;
END PROCESS P_DUPY;

---

PROCESS: P_DUTTY
----------
Purpose:
Inputs: CLE, TRMPTY, CLE, ADST
Outputs: QUADRTTY
The Figure 1.5, the previously description and the Table ?? are a fundamental description that should exist in any case before one starts coding. It eases VHDL-coding drastically, clarifies the data flow and gives always a brief overview of the module.

Sometimes in the previous files we’ve included our own package. This could look like the following one. In the package we declare constants, functions and procedures that are frequently used, whilst their code remains in the package.body section. In our example there are only constants, which can be used either for HDL-coding or for simulation. Note that the type time must be hidden from the synthesis tools, therefore use the Synopsys synthesis off and on switches (they are special comments treated separately) as illustrated.
Now analyze the files with the command:

```
%vhdlan -c -spc exu_pkg.vhd exu_mdmux_ent.vhd exu_mdmux_be.vhd exu_body_ent.vhd exu_body_be.vhd exu_inter_vhd exu_inter_be.vhd exu_inter_be.cfg.vhd
```

The first switch avoids printing a header to the standard output, and the second (-spc) forces the analyzer to check the code for fragments that wouldn’t be syntesizable. When all errors are diminished, proceed with simulation.
begin 
   CLK <= '1';
   wait for t_CLOCK_CYCLE_PERIOD;
   CLK <= '0';
   wait for t_CLOCK_CYCLE_PERIOD;
   end process P_CLK;

P_ADDR_PROCESS
variable r_line: line;
variable control: character;
variable ADDR: std_logic_vector(1 downto 0);
variable w: boolean;
begin
   w<=true;
   while w loop
      if endfile(file) then
         assert false report "File at address: End";
         w<=false;
      end if;
      r_line<=readline(file, r_line);
      readr_line, control;
      if (control='A' or control='B') then
         readr_line, ADDR;
         if ADDR='0' then
            ADDR <= '0';
            wait for (t_CLOCK_CYCLE+1) cycle_time;
            ADDR <= '1';
            wait for (t_CLOCK_CYCLE+1) cycle_time;
         else
            ADDR <= '0';
            wait for (t_CLOCK_CYCLE+1) cycle_time;
         end if;
      end if;
   end loop;
   w<=false;
end process P_ADDR;

P_MMAB Process
variable r_line: line;
variable control: character;
variable crwb and logic;
variable w: boolean;
begin
   w<=true;
   while w loop
      if endfile(file) then
         assert false report "File at address: End";
         w<=false;
      end if;
      r_line<=readline(file, r_line);
      readr_line, control;
      if (control='A') then
         readr_line, crwb;
         if crwb='0' then
            crwb <= '0';
            wait for (t_CLOCK_CYCLE+1) cycle_time;
            crwb <= '1';
            wait for (t_CLOCK_CYCLE+1) cycle_time;
         else
            crwb <= '0';
            wait for (t_CLOCK_CYCLE+1) cycle_time;
         end if;
      end if;
   end loop;
   w<=false;
end process P_MMAB;

P_DATA_PROCESS
variable r_line: line;
variable control: character;
variable data: std_logic_vector(31 downto 0);
variable w: boolean;
begin
   w<=true;
   while w loop
      if endfile(file) then
         assert false report "File at address: End";
         w<=false;
      end if;
      r_line<=readline(file, r_line);
      readr_line, control;
      if (control='0') then
         readr_line, r_line;
         if r_line='0' then
            r_line <= '0';
            wait for (t_CLOCK_CYCLE+1) cycle_time;
            r_line <= '1';
            wait for (t_CLOCK_CYCLE+1) cycle_time;
         else
            r_line <= '0';
            wait for (t_CLOCK_CYCLE+1) cycle_time;
         end if;
      end if;
   end loop;
   w<=false;
end process P_DATA;
Notes

P_RESET_PROCEDURE
variable r_line: line;
variable control: character;
variable ch: std_logic_vector(3 downto 0);
variable w: boolean;
begin
w:=true;
while w loop
if enufilce(3) then
assert false report "Testfile zu Ende";
wait;
end if;
readline(3, r_line);
read_line(3, control);
if (control='A') then
readline(3, r_line);
read_line(3, ch);
if ch = "3100" then
wait for (CLK_period-100 ns setup);
else wait for 10 ns;
wait for 10 ns;
wait for 10 ns;
wait for 10 ns;
end if;
end loop;
wait;
end process P_RESET;

P_BAFFNER_PROCEDURE
variable r_line: line;
variable control: character;
variable chmax: std_logic_vector(1 downto 0);
variable w: boolean;
begin
chmax <= "01";
while w loop
if enufilce(3) then
assert false report "Testfile zu Ende";
wait;
end if;
readline(3, r_line);
read_line(3, control);
if (control='E') then
readline(3, r_line);
read_line(3, chmax);
if chmax = "3100" then
wait for 3*CLK_period;
chmax <= "01";
wait for 10 ns;
wait for 10 ns;
wait for 10 ns;
else readline(3, r_line);
end if;
end loop;
wait;
end process P_BAFFNER;

P_INPUT_PROCEDURE
variable r_line: line;
variable control: character;
variable chmax: std_logic_vector(3 downto 0);
begin

variable w: boolean;

begin
  w := true;
  while w loop
    if modfile(file) then
      assert false report "Errorfile.orig";
      w := false;
      wait;
    end if;
    readline(file, r_line);
    readr_line, control;
    if (control="Y") then
      readline(file, r_line);
      hreadc_line, charvar;
      wait for 3 ms;
      INTERRUPT := charvar;
      wait for 3_t_CLK period + 3 ms;
      wait for t_WAITCYCLE;
    else
      readline(file, r_line);
      end if;
    end loop;
    wait;
    end process P_INTERRUPT;

P_TRUNCATE:PROCESS

variable r_line: line;
variable control: character;
variable charvar: end_logic;
variable w: boolean;

begin
  w := true;
  while w loop
    if modfile(file) then
      assert false report "Errorfile.orig";
      w := false;
      wait;
    end if;
    readline(file, r_line);
    readr_line, control;
    if (control="Y") then
      readline(file, r_line);
      hreadc_line, charvar;
      wait for 3_t_CLK period + 3 ms;
      wait for t_WAITCYCLE;
      null;
    end if;
    readline(file, r_line);
    end if;
    end loop;
    wait;
    end process P_TRUNCATE;

P_TRUNCATE:PROCESS

variable r_line: line;
variable control: character;
variable charvar: end_logic;
variable w: boolean;

begin
  w := true;
  while w loop
    if modfile(file) then
      assert false report "Errorfile.orig";
      w := false;
      wait;
    end if;
    readline(file, r_line);
    readr_line, control;
    if (control="Y") then
      readline(file, r_line);
      hreadc_line, charvar;
      wait for 3_t_CLK period + 3 ms;
      wait for t_WAITCYCLE;
      null;
    end if;
    readline(file, r_line);
    end if;
    end loop;
    wait;
    end process P_TRUNCATE;

end th;

configuration cfg_ax1_tb of ax1_tb is
  for tb
    for sample unit:
      wee see entity work.EU_BEHAVIORAL;
    end for;
  end for;
end cfg_ax1_tb;

The testbench that is illustrated now should explain a way of coding to make testing a bit more modular and easy to use. It is based on some preassumptions, that when they
are met by the design, make it easy to use the testbench with little modifications on all different kinds of modules. The assumption is that you have a periodic bus-cycle and that all signals that don't apply to such a category can be modeled in the same periodic fashion. What follows is a testbench file that reads stimuli from an ASCII data file. According to the periodic signals, the stimuli are taken from this file only, when they should be valid. In the gaps between, when the values are of no interest for the bus-cycle, they are set to a default value, that is usually related to the idle time. The testbench is structured into several processes, where every process reads simultaneously to all others the stimuli data. The data file consists of a comment line followed by the data. The relevant data for a process checks, if the first letter in a comment line corresponds to the letter that is applied to it. If the comparison is true the following data in the next line is applied to the described models. Therefore to keep all processes simultaneous, within every bus-cycle must exist a data value for every process.
Due to the fact that simulation is often a very interactive process, create an include file that holds the relevant commands for the simulator (e.g. which signals to trace). The following could be an example for the previously described EXU.
Now analyze the testbench with the command

```
%vhdlan -nc exu_tb.vhd
```

and start the simulator either via

```
%vhdlbx -i exu_tb.inc CFG_EXU_TB
%vhdlx -i exu_tb.inc CFG_EXU_TB.
```

In addition to the commands specified in the include file, you can enter commands at the command prompt (e.g. `run 3000`). The waveform window (see Figure 1.8) is rather intuitive.

When the simulation runs perform as intended, you can continue with synthesis. The major functionalities for synthesis were already described in the prior sections. Therefore, the commands of the following script should already be at hand.
The first part of this script analyzes, constrains and synthesizes the example unit (EXU). Refer to previous sections for an explanation. Afterwards scan-path-insertion is performed. First the scan methodology is set to full-scan and the scan-style is set to multiplexed flip flop. The next few lines define the timing related test attributes given by the ES2 documents. The check_test command is useful to identify non-scannable sequential cells. Then with the command insert_test scan-cells are inserted and the scan-path is instantiated. Afterwards the test select input (test_se) is constrained and buffers are instantiated to fix design-rule violations. The create_test_patterns instruction calculates test patterns for this path. Next the reports give an overview of the derived results.

Before the modules are made ready for transfer to Cadence, pads should be instantiated. One way would be to write a structural VHDL-source file, that instantiates several VERILOG-files after this stage, that can be read into CAdence via the Verilog-In procedure. –Refer to the next section how to continue then with the layout.– Now before you proceed with Cadence, you should perform a post-synthesis simulation. – Take the last saved design unit (in our case scan/exu_scanned.db) and load it into de_shell:

```bash
>read -f db scan/exu_scanned.db
>current_design exu_body.test.1
>ungroup -all
>current_design exu
>ungroup -all
>write -f vhdl -o scan/exu_synthesized.vhd
```

Then edit this VHDL-file, to check that the lines for the library inclusion are correct (e.g. use work.ECPD07.all), and the name of the instantiated architecture (e.g. SYN_BEHAVIORAL). In general both items are handled via the setup-files and should be already instantiated as intended.

In addition you have to edit your testbench file now to add a process that drives the test-select and test-input lines, the component to match the entity with the newly added test-ports, the port-mapping to the component and finally the configuration at the end of the testbench to point to the correct architecture.

Analyze the VHDL-netlist exu_synthesized.vhd and the modified testbench and invoke the simulator the same way you did for simulation at the logic level.
Chapter 2

Back-End design with CADENCE

Before you start with the back-end design of your project using CADENCE, make sure you got an own directory from within your project directory available (e.g. /project/<projectname>/cds). You can make a soft-link from within your home-directory to ease traversing through the system. Furthermore you need to have two files (or two links) in this directory called .cdsinit and .stimc. All this should already have been done by your project head, if not contact him.

Before invoking Cadence you must allow the program to display the windows on your terminal. Therefore type % xhost [terminalname] within an xterm window. Change to the working directory (e.g. /project/<projectname>/cds) and enter % icfb &. The command interface window (CIW) is displayed as illustrated in figure 2.1.

![Figure 2.1: The CIW interface](image)

First create a new library that corresponds to the name of your project (e.g. exu) with Open → Library ←. In the displayed form fill in: Library Name [exu] ←. You are now prompted to create a new library - select ←. Then in the next form specify a working and/or user group and apply the according permissions, leave all other items to their default. Open the library browser Design Manager → Library Browser you should see your newly created library.

At the CIW command line prompt enter load "/project/proto/VerilogInit.il". The form for importing Verilog files, with some defaults preset, is displayed in table 2.1. From several items check and adjust the following settings:

The import session can take several minutes up to several hours - stay patient. - The CIW reports many Warnings and Errors that should be checked carefully. - After the import process has finished you should have at least one schematic view in your design library, which you should open (e.g. via the Library Browser). Within the schematic view of your top-level design perform a Check and Save in every level of the hierarchy in a
Target Library Name: exu
Reference Libraries: StdLib PadLib basic
Verilog Design Files: /project/exu/syn/s2c
Verilog Options: /users/ict2/staff/es2lib/ecpd07/utl/verilog/dlib.ind

Verilog Cell Modules:
- View
- Import
- Schematic

Table 2.1: The VerilogIn form

bottom up manner. Before doing so at the top instance, perform the last final changes as instantiating and connecting an oscillator cell, a power-on reset and all the other elements that make up the final design. Furthermore you require for e.g. a cell called LIBTOPNETS, that adds information for the supply nets, and some pads for both supply of core and peripheral. How many and what kind of pads have to be used therefore is described in [ES294] on page 3-3 ff.

Figure 2.2: The Schematic View

When you have fixed everything and done the final 'check and save' operation, bring up the hierarchy browser via Tools → Floorplan/Schematics and Floorplan → Hierarchy Browser. For a flat design, click on the top instance and perform Hierarchy → Generate Physical Hierarchy. The views autoLayout and autoAbstract are created.

For a hierarchical design, the above action should be repeated for every level of hierarchy - therefore expand the top element by instance. Alternatively you can also open the flat autoLayout and use the browser to cross select the instances of each hierarchy and perform a Create → Softblock. Anyway don't regroup the hierarchy due to the fact that CADENCE would rename several nets. Then you would no longer be able to run an LVS check later on.

The previously generated autoLayout can now be opened into Cell Ensemble by selecting Tools → Floorplan/P&R → Cell Ensemble. As a first step load the net information for routing (track widths and separation of distinct nets).
Route → Modify Net → Net Properties File
♦ users/ict2/staff/es2lib/ecpd07/util/startup/netData
♦ read

This file contains information for the nets vdd1 and gnd1. Now add glue cells such as LIBCORNER with Place → Glue Cell → Add. To move the cells out of the placement region perform Floorplan → Reinitialize. Then make Region selectable in the Object Selection Window (OSW) and click on the default region, that was created after reinitialisation. Perform Floorplan → Analyze Floorplan Objects and enter the desired number of rows (use an even number preferably) and a percentage for the desired row utilization. Both values require some experience and must be derived in an interactive manner. (e.g. 90% row utilization) – Keep in mind that the router expands primarily up and a bit on both sides but not down! – Drag the placement region in a way that the routing channels between the standard cell rows become a bit smaller in height than the standard cells. Therefore considering these aspects, the width of the placement region should be greater than the height in case you have a square core area in mind. Now adjust the design outline to the default region and perform a second reinitialisation with only the instance status left on.

Load the floorplan file Place → I/O commands → Read Initial File to place the pads around the placement region. When the file is correct all but the LIBCORNER pads should be moved to their desired places. In some cases the placer swaps a few pads, therefore control the placement carefully. Than move [m] and rotate [F3] the corner cells to their locations and use the command Edit → Align Cells to position them and change the property \[q\]\ from unplaced to placed.

| VDDPY1 | left  | 1 |
| PIN_TEST_SE | left  | 2 |
| PIN_TEST_SI1 | left  | 3 |
| ... |
| GNDPY1 | bottom | 1 |
| PIN_DATAI_0 | bottom | 2 |
| PIN_DATAI_1 | bottom | 3 |
| ... |
| PIN_CSUTIME13 | right | 23 |
| VDDPY5 | right | 24 |
| GNDPY2 | top | 1 |
| PIN_CSUTIME14 | top | 2 |
| ... |

Start the placement of the standard cells with Place → Automatic

♦ Insert Feedthru

Feedthru Library Name: StdLib
Feedthru Master Name: LIBFEED
Placement Snap Grid: 0.1 (Ecpd07)

Sometimes after placement of the standard cells you have to fix up the position of the corner cells again. Now bring up the menu for the power cells Place → Power Cell → Add Manual. Depending on the size of the placement region you add either only cap cells on each end of a row, or when the length of a row exceeds some values, influenced by the operating frequency (see [ES294] on page 3-6), you'll have to add power bars. For the later take care that the cells have all the same x-positions after placement,
otherwise the routing becomes worse. In the Define Power Cell menu select LIBLCAP and LIBRCAP for the cap cells. For powerbar grid routing add also LIBPGB40 and LIBPGBFILL and disable the distribute feedthru option. Usually the later two should be omitted in most cases. Now draw two rectangles over the left and right side of the standard cell rows and select . Power cells should be inserted, if not correct the placement and repeat the above step. Figure 2.3 gives a glimpse of an exemplary placement.

![Figure 2.3: A placed design](image)

Now select all instances and do a Place → Snap to Grid with a value of 0.1.

Placement is now finished and can be saved as for e.g. placed.

The next step is Route → Channels → Create and to modify the net properties of dedicated nets, such as for e.g. the clock net, via Route → Modify Net → Modify Net Properties. To determine the width and separation of the clock net see [ES294] on page C-6. Set the preferred layer to CME2, the signal type to clock and the criticality to 105.

Then invoke interactive global route on the clock and supply nets. Route → Global Route → Interactive Global Route, select Initialize Net and enter a net name for global routing (e.g. [Clock, vdd!, gnd!]). With Settings you can enlarge the snap points from 1 to say 19. You should recognize all available snap points for the selected net. You can select those you want to connect either by clicking on them or via the Select Scan Chain menu. Take care of the order you select points, because it can influence the desired routing order. When you’ve all desired nets selected, Connect Set will tie them together. Via Modify Set you can apply a routing width and a preferred layer for the selected segments. Therefore if you want different widths of segments (e.g. the main clock trunk is in general far thicker than the clock net within a channel), you’ve to modify those separate parts individually. Before you push the Exit Net button, the modified snap points should be selected, otherwise you can get malicious routing results. On exiting the net, select the Fix Global Route option and to remove all the snap points. If you’re prompted for an edge connected attribute, you’ve obviously
forgotten to connect some instances.

After all special nets are global routed, you can exit the Interactive Global Route menu by hitting the Done button. Now perform global routing Route → Global Route → Automatic ↔. This takes a while - take a look at the CIW. Then continue with Route → Detail Route → Automatic. Choose the Compact button to set the routing grid to 0.1 microns and start the detail router. If errors occur they are prompted in the CIW whilst routing. Then save the design under a meaningful name (e.g. layout), as shown in figure 2.4.

![Figure 2.4: The final layout](image)

Now that placement and routing is finished, finalize the chip by performing a DRC and LVS run followed by a final sign-off simulation.

Open the previously saved view and change to the layout interface Tools → Layout. In the CIW window select Technology File → Compile Technology and specify

```
/users/ict2/staff/es2lib/ecpd07/uttl/startup/diva.PR\$load
```

In the layout view choose Verify → DRC in full and flat mode, with the switches grid and correct set. In the case that errors are reported, you can view them via Verify → Markers → Find.

Tip: In the layout view you can modify segments easily by changing the values of the geometries you get from the properties window. Modification takes place on a solely segmented basis, therefore if you move a segment it is disconnected from the rest. In contrast the cell ensemble view remains the segments tied to each other when modified.

If you’re not able to select a wire, you must first descend into the desired module/channel and perform the check there once more to locate the same errors again. If you want to circumfer the required steps of traversing through the hierarchy, you can explode the channels from within the cell-ensemble view by Route → Detail Route → Explode Channels, which will promote the channels to the upper hierarchy level.
In the case when the DRC checker reports zero faults proceed with **Verify → Extract** and select **Macro Cell** as extract method. When succeeding a view called extracted is created, that is used for the LVS run.

Prior to the LVS run compile the technology file `diva.lvs`, that resides under the same directory as the one before for the DRC run. Then start **Verify → LVS** to compare the schematic versus the extracted view. In the case the run succeeds check the log file if the two views match; - if not search the directories LVS/schematic and LVS/layout for files with a .out extension (e.g. mergen.out). These files contain information about the nets that caused the mismatch. (e.g. A merged net in the schematic is mostly caused by a short in the layout.) Open the layout in cell ensemble and highlight the conflicting nets with **Edit → Search**. Zoom into the regions where these nets cross and search for violations. Correct them and run the DRC, EXTRACT and LVS again. Hopefully you’ve corrected everything without introducing new errors, and the netlists now match.

- Save the design!

How to perform a post layout simulation? Within the cell ensemble view select all global nets with $[1, N]$ and perform an **Analyze → Paraeastics → Extract** on them. Then make sure that you have enough swap space available, before succeeding with the next step **Analyze → Paraeastics → Write reduced SPF** with the option **physical name mapping**. The name of the SPF file should be `<top.Cellname > .spf`. At the CIW command prompt enter `ES2generateSDF`. This brings up a form, where you have to enter the library name, the name of the top cell and which kind of SDF (min, typ or max) to generate. When you select all three, you’ll end up with six files (three sdf files and three reports with an extension .out). Check the reports carefully whether you’re violating fanout and fan-in rules or not. If you got violations cross check them with the library data book.

![Note: If you connect two clock buffers in parallel and the load is greater than the one a buffer can drive, you’ll see a warning. Check if the given load is less than the added load of the buffers.](image)

Now let’s start with simulation. Open from within the schematic via **Tools → Simulation** the Verilog-XL environment. Change **Setup → Record Signals** from `Top Level I/O` to `All Signals`. Enter at the CIW prompt `ES2simTemplate` and leave all options to their default values. Change to the simulation run-directory and edit the generated file `ES2testfixture.v`. At the bottom of the file you’ll see a line for inclusion of an SDF file `top.sdf` and an inclusion of another verilog file (default.sim.v) that supplies the user provided stimuli. An example of such a stimuli file is provided in appendix ???. Change the name to point to your file. What is still missing, is a file top.sdf in the simulation directory that has valid back-annotation information. Therefore enter at the CIW prompt `ES2sdfTranslation`. This copies the appropriate SDF file into the directory, generates a report that should be checked, and produces a soft-link called top.sdf that points to the real SDF file.

![TIP: If you have connected two clock buffers in parallel, check that the back-annotated SDF values for the buffer delays don’t differ too much, otherwise your simulation run won’t be successful. Make the two delays for CKBUF1 and CKBUF2 equal.](image)

Now press the **Start Interactive** button and simulation will start unless you haven’t introduced any error in a previous stage. When simulation ends bring up the waveform viewer via **Debug → Utilities → View Waveform**. The simulation results that are stored below `simDir/shm/db` in the run-directory is loaded automatically. You can now add signals of different levels to the waveform window by **Edit → Browser/Display Tool**. When all desired signals are on screen save the setup for other/later sessions. After
you've checked the simulation (see figure 1.8) rename the directory *shm.db* and edit the previously saved setup-file to point there. This will avoid that the simulation run can be overwritten by follow-up runs.

![Image](image.png)

**Figure 2.5:** The Waveform window

When everything 's fine you can create a *gds2* file out of the final layout view via **Translators → Stream Out** from the CIW. Fill in the working directory, choose Stream DB, Library Version 5.0, microns, append a layer map table (see appendix ?? for details) and activate in the options form the items report bad polygons and precision report. In the end you can load the script `>load "/project/proto/retr.il"`. Then evaluate these script via `>eval (retr())` to get some statistics and reports of the final layout.
Bibliography


