Power Analysis with Quartus II and Modelsim

Sabih H. Gerez
Chair of Computer Architecture and Embedded Systems
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente

September 15, 2009

1 Preliminaries

This document illustrates how a design specified in VHDL can be analyzed for its power consumption using Altera Quartus II and assisted by Modelsim. After having studied this document and performed the exercises, the reader should be able to estimate the power consumption of her own design.

The document is based on the following versions of the software:

- Quartus II Version 9.0
- Modelsim 6.4a

It is assumed that the reader is familiar with the following document:

[1] E. Molenkamp, "*VHDL Tutorial*", University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science, August 2009.

2 Introduction to Power Estimation

As explained in the oral lecture, there are many ways to perform power estimations. The more information is available, the more accurate the estimation becomes. The method presented in this document is one of the most accurate methods that can be performed without resorting to analog simulation. It consists of the following steps, most of which are separately explained in the next sections:

- Simulating the given design at the register transfer (RT) level. This step does not play a role in the estimation itself, but is essential to understand the testbench and to ascertain that the circuit behaves as intended.
- Synthesizing the design. After successful synthesis, Quartus will have generated a post-synthesis description of the design in VHDL and an SDF file containing information on delays. Quartus will also generate a script to record all signal activities in a post-synthesis simulation.
- Performing a post-synthesis simulation. The VHDL, SDF and script are simulated in Modelsim. The result of the simulation is a value-change dump file (VCD) that contains all the signal toggles that occurred during simulation.
- The VCD file is read by Quartus. The switching activities for each signal are extracted from the VCD file. After processing the information and combining it with models on the FPGA used, a power consumption report is generated.

3 Vehicle Design and its Pre-Synthesis Simulation

The vehicle design that will be used to illustrate the power analysis method is an *electronic die*. The intended function of this circuit is that a pseudo-random number in the range 1 to 6 (corresponding to one of the six side surfaces of a cube) should be displayed when a button is pressed. The behavior of the display when the button is not pressed, is unspecified.

Its VHDL entity declaration is given in Figure 1. Apart from the *reset* and *clock* signals that every synchronous circuit has, it has two interface signals: the logic signal `button` indicates whether the

button is pressed (`'1'`) or depressed (`'0'`); the 7-bit logic signal `display` tells which of the segments of a seven-segment display should be turned on to display the thrown dice value.

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

entity die is
  port (clk     : in std_logic;
        reset   : in std_logic;
        button  : in std_logic;
        display : out std_logic_vector(0 to 6));
end die;
```

Figure 1. *The declaration of the die entity as found in file* `die_ent.vhd`.

One way to implement the intended behavior is given by the architecture `cont` shown in Figure 2. The hardware has a counter `die_value` that counts at a much faster speed than the frequency at which the button will be pressed. The architecture name `cont` refers to the continuous copying of the counter value to the display. Only when the button is pressed, the copying is suspended and the display shows a number that can be considered random. The numbering convention of the display segments is shown in comments in the figure.

Unzip the file `dds-power.zip` and then compile the following files in Modelsim in the given order (start in a clean directory with a new `work` library, consult [1] if you are not yet familiar with Modelsim):

- `die_ent.vhd`: This file contains the entity declaration that has already been discussed.
- `die_cont_arch.vhd`: This file contains the cont architecture declaration that has already been discussed.
- `tvc_die.vhd`: This file contains the *test-vector controller* (TVC). It provides the stimuli for the *design under verification* (DUV), `die`. It generates the reset signal, a clock signal of 50 MHz and button pressing at an approximate frequency of 100 kHz. The button is pressed at about 5% of the time. Note that this frequency is orders of magnitude higher than a human being could handle; however, the simulation times would become very long if the button pressing was performed with intervals of seconds. The clock frequency of 50 MHz was chosen to be realistic for a current-day design and to be able to distinguish some dynamic power consumption next to the static power (leakage currents). The test-vector controller also has a process that translates the display signals back into an integer. By monitoring signal `thrown_value` in Modelsim, one can easily see which die value was generated.
- `tb_die.vhd`: This is the testbench file. It connects the DUV with the TVC and as such the top-level entity in the simulator.
- `conf_tb_die_cont.vhd`: This is the *configuration* to be simulated. For those who are not familiar with VHDL configurations: a configuration links architectures to entities in the entire design hierarchy. It is especially useful when multiple architectures exist for the same entity, as will become clear later on.

```
-- 7-segment display coding
--
--      (0)
--     +---+
--     |   |
-- (5)|   |(1)
--     |(6)|
--     +---+
--     |   |
-- (4)|   |(2)
--     |(3)|
--     +---+

architecture cont of die is
  signal die_value : integer range 0 to 5;
begin
  process(reset,clk)
  begin
    if reset='1' then
      die_value <= 1;
      display <= "1111110"; -- display a 0 at reset
    elsif rising_edge(clk) then
      if die_value<5 then
        die_value <= die_value + 1;
      else
        die_value <= 0;
      end if;
      if button='0' then
        case die_value is    --0123456
          when 0 => display <= "0110000"; -- display a 1
          when 1 => display <= "1101101"; -- display a 2
          when 2 => display <= "1111001"; -- display a 3
          when 3 => display <= "0110011"; -- display a 4
          when 4 => display <= "1011011"; -- display a 5
          when 5 => display <= "1011111"; -- display a 6
        end case;
      end if;
    end if;
  end process;
end cont;
```

Figure 2. *The* `cont` *architecture of entity* `die`*, as given in file* `die_cont_arch.vhd`.

After compiling the files, simulate the configuration `conf_tb_die_cont`. Select the relevant signals to display in the Wave window of Modelsim. Run the simulation for 200 us, which should simulate some 20 "throws" of the die. Do you see that the display is continuously changing value except when the button is pressed?

4 Synthesis with Quartus

The circuit that you have just simulated at the RT level will now be synthesized in Quartus. Follow the instructions in [1] to perform the synthesis, paying attention to the following:

- Use `die` as the project name, as well as the name of the top-level design entity.

- Add the files `die_ent.vhd` and `die_cont_arch.vhd` as the design files.
- Select the same device as in [1], the Cyclone EP1C12F324C8.

Before starting the synthesis process, a few more things should be set: Invoke the simulation settings pop-up dialog by: Assignments → EDA Tool Settings → Simulation and check the square for generating a VCD file as indicated in Figure 3. Provide also the design instance name, which should be the top-level instance name of your hardware (without the testbench), in this case: `duv`.

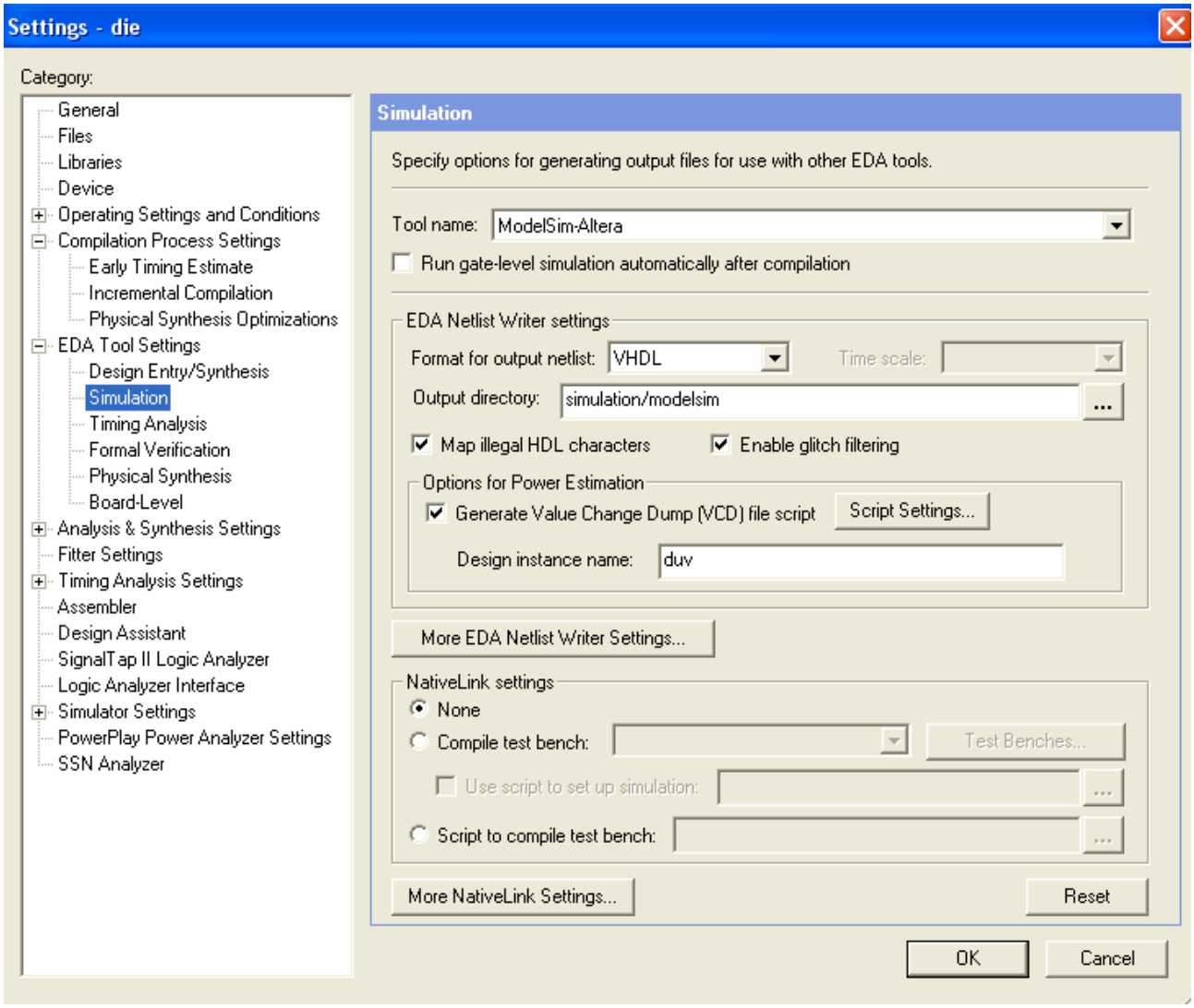

Figure 3. *The Quartus simulation-settings pop-up window.*

Now go on to set more parameters by clicking on "More EDA Netlist Writer Settings". Change the value of the "architecture name in VHDL output netlist" to `cont_post` and turn ON the "do not write top level VHDL entity" flag. The architecture name is used in the testbench VHDL configuration for post-synthesis simulation. It is not necessary to generate the VHDL entity as it does not change during synthesis. The pop-up window for these settings is shown in Figure 4.

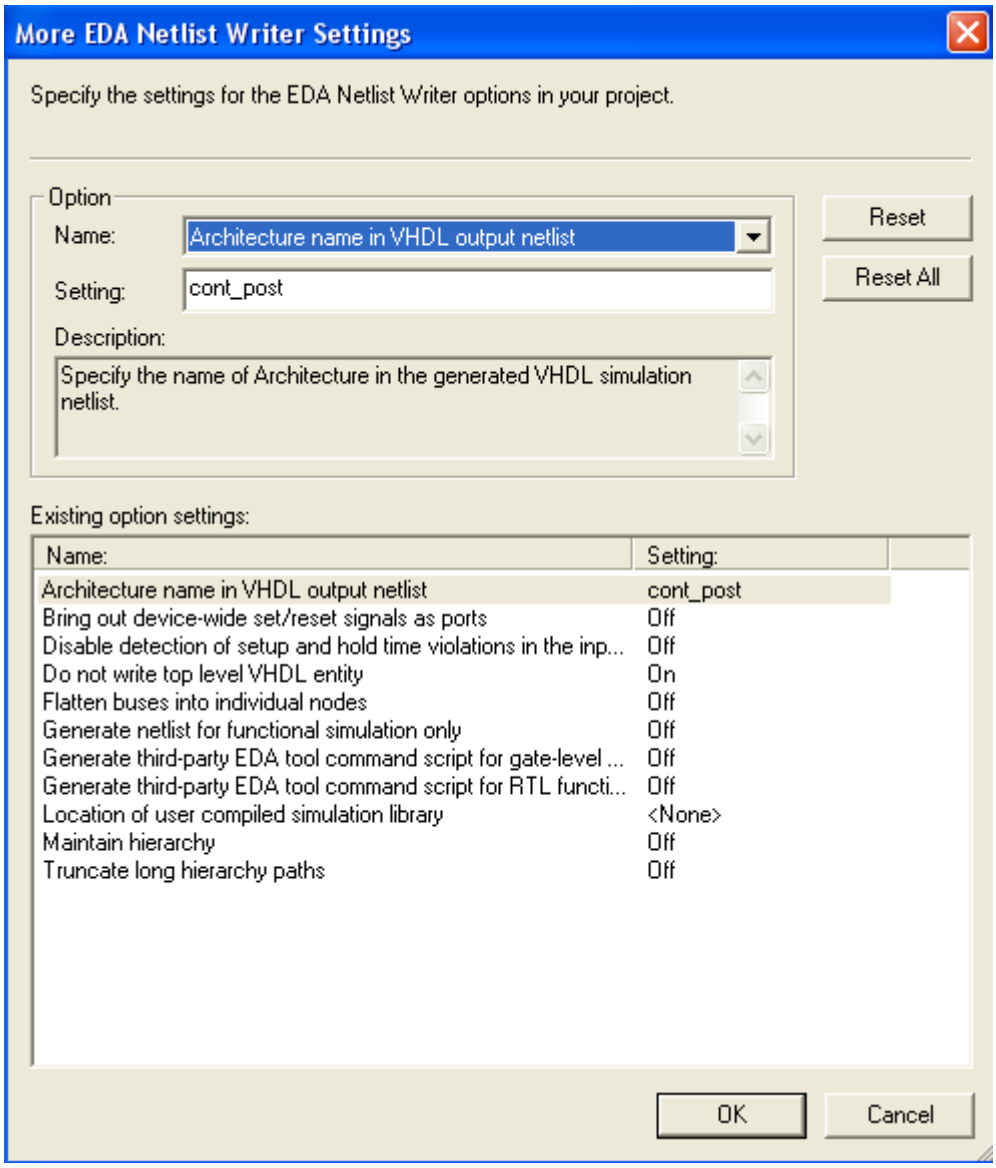

Figure 4. *Advanced Quartus settings for the netlist writer.*

With these settings you can execute synthesis: Processing → Start Compilation. After some time, the compilation will finish. Warnings are generated that can be likely ignored. The results that matter for the purpose of power analysis are three files to be found in the subdirectory `simulation/modelsim` (w.r.t. the working directory of Quartus):

- `die.vho`: the gate-level netlist in VHDL of the synthesized design.
- `die_vhd.sdo`: the SDF file associated to this netlist.
- `die_dump_all_vcd_nodes.tcl`: a script file to record all signals during simulation.

5 Post-Synthesis Simulation

Although Quartus has the possibility to launch a Modelsim simulation, the post-synthesis simulation of the design will be performed directly from Modelsim, from the same environment as for the

pre-synthesis simulation (so using the same `work` library). Compile the following two files into the old library `work`:

- `die.vho`: the result of synthesis.
- `conf_tb_die_cont_post.vhd`: the configuration that plugs in the `cont_post` architecture of die into the testbench.

Now start the simulation of the configuration `conf_tb_die_cont_post`; add the SDF file as described in [1] and fill `/duv` in the "Apply to Region" field. Modelsim will load the design and should report "SDF Backannotation Successfully Completed" before showing the VSIM prompt. Think of using picosecond resolution.

Before actually running the simulation, load the script `die_dump_all_vcd_nodes.tcl` that is generated by Quartus, as mentioned above. You can load the script with File → Load. You can also select the signals that you want to observe in the Wave window.

Now run the simulation for 200 us as for the pre-synthesis case. You should see similar results as in the pre-synthesis simulation. However, zooming in on the time axis should reveal that there is a delay between the clock edge and the time that outputs change value. In addition, not all outputs should switch simultaneously.

Quit Modelsim. If everything went well, there should be a file `die.vcd` in the directory from which Modelsim was operated. This file contains all switching activity and will be used by Quartus to generate a power analysis report.

6 Power Analysis

The information for performing power analysis is now available. It will be done by the Quartus tool PowerPlay. If you had quit Quartus, restart it and load project `die.qpf`.

Before invoking PowerPlay, some parameters need to be set. Open the correct pop-up window by Assignments → Settings → PowerPlay Power Analyzer Settings as shown in Figure 5. Add `die.vcd` as the input file for deriving toggle rates. Check the fields "Write signal activities to report file" and "Write power dissipation by block to report file". You do not need to touch the default toggle rates; the should be ignored as all actual rates are already in `die.vcd`.

You can now perform the analysis with Processing -> PowerPlay Power Analysis Tool. The resulting reports give you detailed information on the power consumption. The summary is the most important one, but you can also get information about subblocks of the design and individual signals.

In the case of the die circuit, it turns out that the static power dissipation (leakage) is dominant. This is not surprising as only a very small part of the FPGA is used. The power dissipated in the I/O is significant. Write down the results summary for the purpose of future comparison with an alternative implementation of the electronic die.

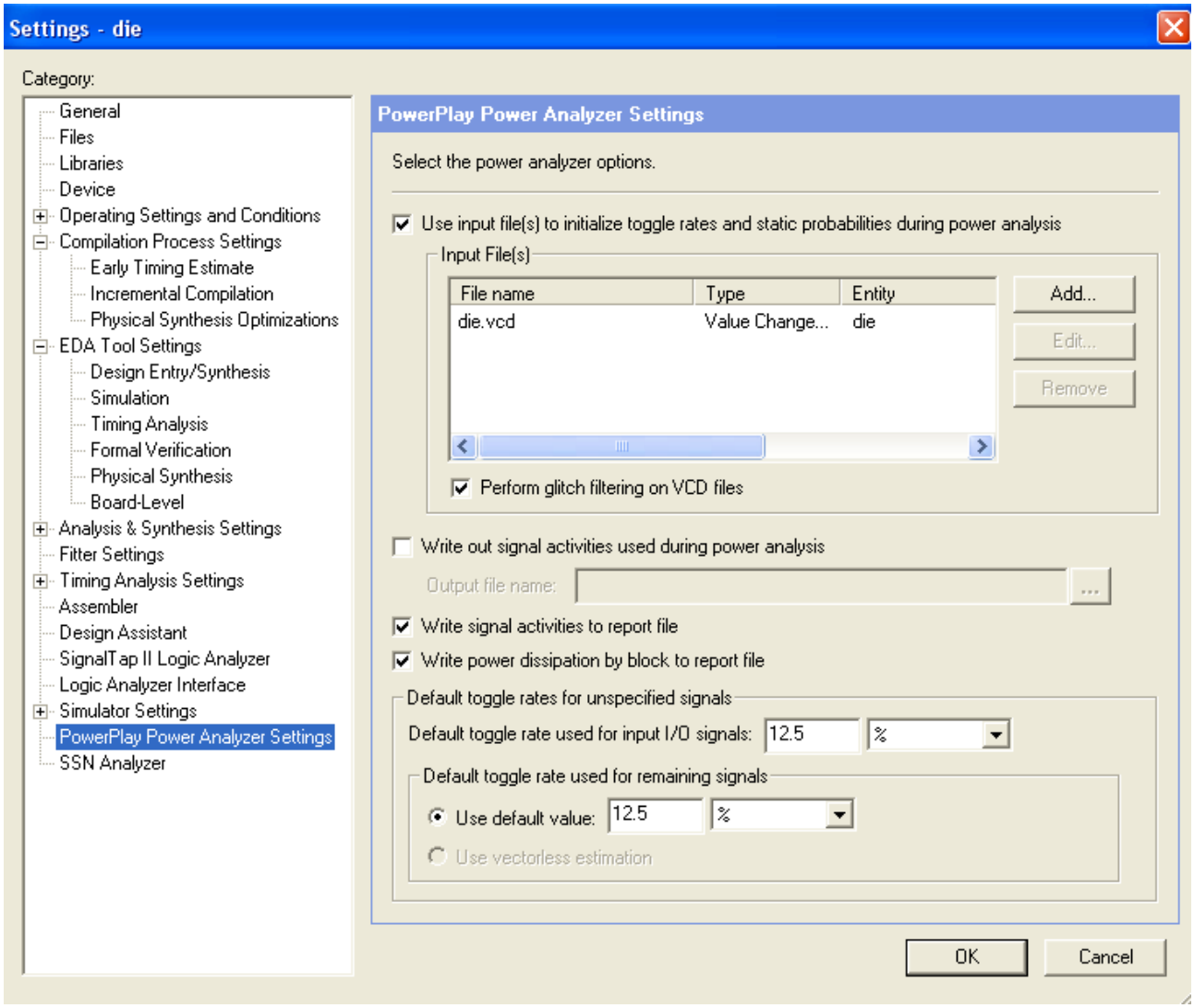

Figure 5. *Settings pop-up for configuration of PowerPlay.*

7 Alternative Design

The conclusion of the power analysis of the `cont` architecture of `die` is that quite some power is dissipated in the I/O drivers. Obviously, this can be attributed to the fact that the internal counter value is connected to the display when the button is not pressed, resulting in lots of unnecessary signal transitions. An alternative architecture for the die circuit, called `gated`, is shown in Figure 6. Instead of continuously passing the counter value to the display, it only updates the display when the button is pressed, on the rising edge of the signal button. This requires one extra flipflop in the hardware, `button_prev`, that remembers the previous value of button (note that the button is considered ideal and no debouncing is implemented).

Repeat all design steps performed for the cont architecture for the gated architecture:

- First perform the pre-synthesis simulation by compiling the files `die_gated_arch.vhd` and `conf_tb_die_gated.vhd` into the existing work library and then simulating configuration `conf_tb_die_gated`. Do you notice the different behavior of the display?

- Then synthesize the new design. As opposed to Modelsim, Quartus cannot easily deal with multiple architectures for the same entity. The most practical approach is to take the previous project and replace `die_cont_arch.vhd` by `die_gated_arch.vhd` in the list of files for the project. Before running synthesis, change the architecture name in the VHDL output netlist from `cont_post` to `gated_post`. Running synthesis will overwrite the output files `die.vho` and `die_vhd.sdo`.

- Recompile `die.vho` and then compile `conf_tb_die_gated_post.vhd`. Perform a post-synthesis simulation as described above.

- Finally, rerun PowerPlay. How did the I/O power change? And how the dynamic power? The static power should have remain unchanged. Explain your results.

```
architecture gated of die is
  signal button_prev: std_logic;
  signal die_value: integer range 0 to 5;
begin
  process(reset,clk)
  begin
    if reset='1' then
      die_value <= 1;
      button_prev <= '0';
      display <= "1111110"; -- display a 0 at reset
    elsif rising_edge(clk) then
      if die_value < 5 then
        die_value <= die_value + 1;
      else
        die_value <= 0;
      end if;

      -- remember button value for next iteration
      button_prev <= button;

      -- only update display on rising edge of "button
      if button='1' and button_prev ='0' then
        case die_value is    --0123456
          when 0 => display <= "0110000";
          when 1 => display <= "1101101";
          when 2 => display <= "1111001";
          when 3 => display <= "0110011";
          when 4 => display <= "1011011";
          when 5 => display <= "1011111";
        end case;
      end if;
    end if;
  end process;
end gated;
```

Figure 6. *An alternative architecture called* `gated` *for* `die` *as given in file* `die_gated_arch.vhd`.