# *Validating the CPU in Prototyping Hardware*

In this step, we will test the CPU and application, which were generated in the previous steps, on a FPGA prototyping system. For this purpose, we will use the DE2-115 Proto Board from ALTERA shown in . It contains a Cyclone IV FPGA chip.

Figure ‑   
DE2-115 Prototyping Board from Altera.

## Creating the Quartus Project

Quartus is a program from Altera to support the whole tool-flow from managing your source files over synthesis, map, and place & route until finally uploading the design to the FPGA board.

If your local workstation is rather slow (or it does not have the Quartus tool), then you should instead connect to a faster pc (e.g. i80pc85, i80pc86, and i80pc60) and run Quartus there: ssh –X i80pcXX.

Start Quartus by just executing *quartus*

If you do not already have a project for your current CPU, then create a new one: “File” – “a New Project (New Project Wizard)”. As Project Location you should choose your ASIP Meister Project Directory (e.g. ASIPMeisterProjects/Session5/Quartus-basis) and as Project Name you can choose something like “cpu-basis”. You can ignore the upcoming window “Add Files” at this moment as you will do this step later. In the upcoming window “Family & Device Settings”, you have to adjust the values to the data shown as follows: (Family: Cyclone IV E, Name: EP4CE115F29C7) see . Afterwards just press “Next” – “Next” – “Finish” to create an empty project for your CPU. The device settings are needed to make sure, that the map, place and route tools know exactly the type of the target FPGA.

Now you have to add the needed VHDL and constraint files to your Quartus project, by right clicking on the files (at the upper left inside your Quartus Project Navigator Window, see ) and choosing “Add/Remove files in Projects”.

There are three types of files that are needed for a hardware implementation:

* CPU VHDL Files
* Framework Files and IP cores

**CPU VHDL Files** have been generated using ASIP Meister and they can be found in the meister/{CPU-Name}.syn directory, as explained in Chapter 2.3.

**Framework Files** are important for the connection between CPU, Memory, Audio, and all other components. They are predesigned for this laboratory and they are available in ~asip00/­ASIPMeisterProjects/­TEMPLATE\_PROJECT/­Quartus\_Framework. The framework consists of the following three types of file and all of them have to be added to the Quartus project.

* The VHDL files describe how all components are connected together.
* The IP-Cores are used within the framework, e.g. memory blocks for instruction- and data memory or FIFOs for the connection to the Audio. These IP cores are not available as VHDL source code, but instead they are available as pre-synthesized net lists (e.g. bram\_dm.qip, audio\_out\_fifo.qip).
* The SDC (Synopsys Design Constraint) file describes the user constraints (e.g. which clock frequency is requested).
* Pin Location file (PINs.csv) describes the connections between pins in the Toplevel entity and the FPGA pins.

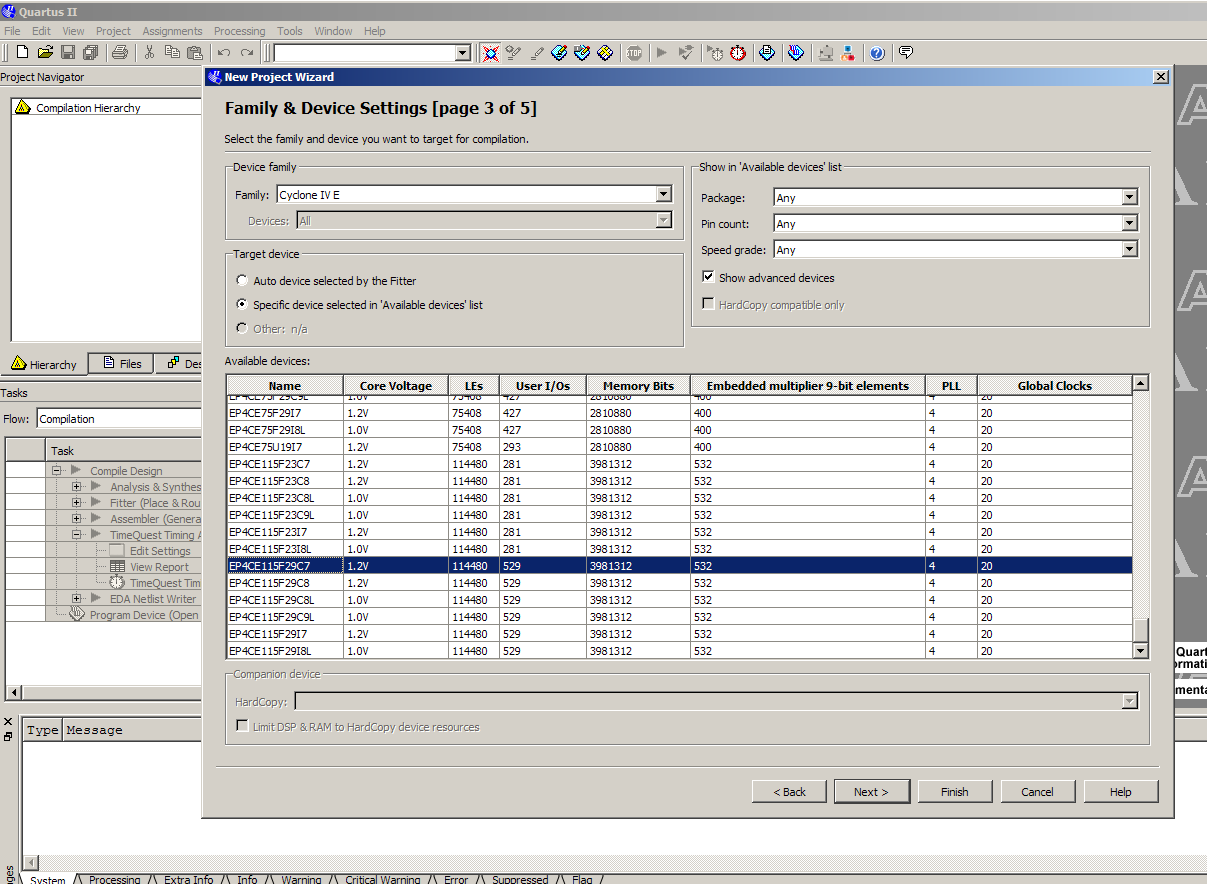


Figure 6‑2. Quartus Device Properties

After you have added all needed files to your Quartus project/directory, you can see in the sources sub window all the source files. It Will be shown how the files are structured, i.e. which file instantiates which other file, just after you Compile your project.

## Synthesizing and implementing the Quartus project

After you have added all needed files to your Quartus project, you have to select which VHDL file is the Toplevel entity for your project (dlx\_toplevel.vhd). You can select a file as a Toplevel entity by right click on it and then click on Set as Top-Level Entity.

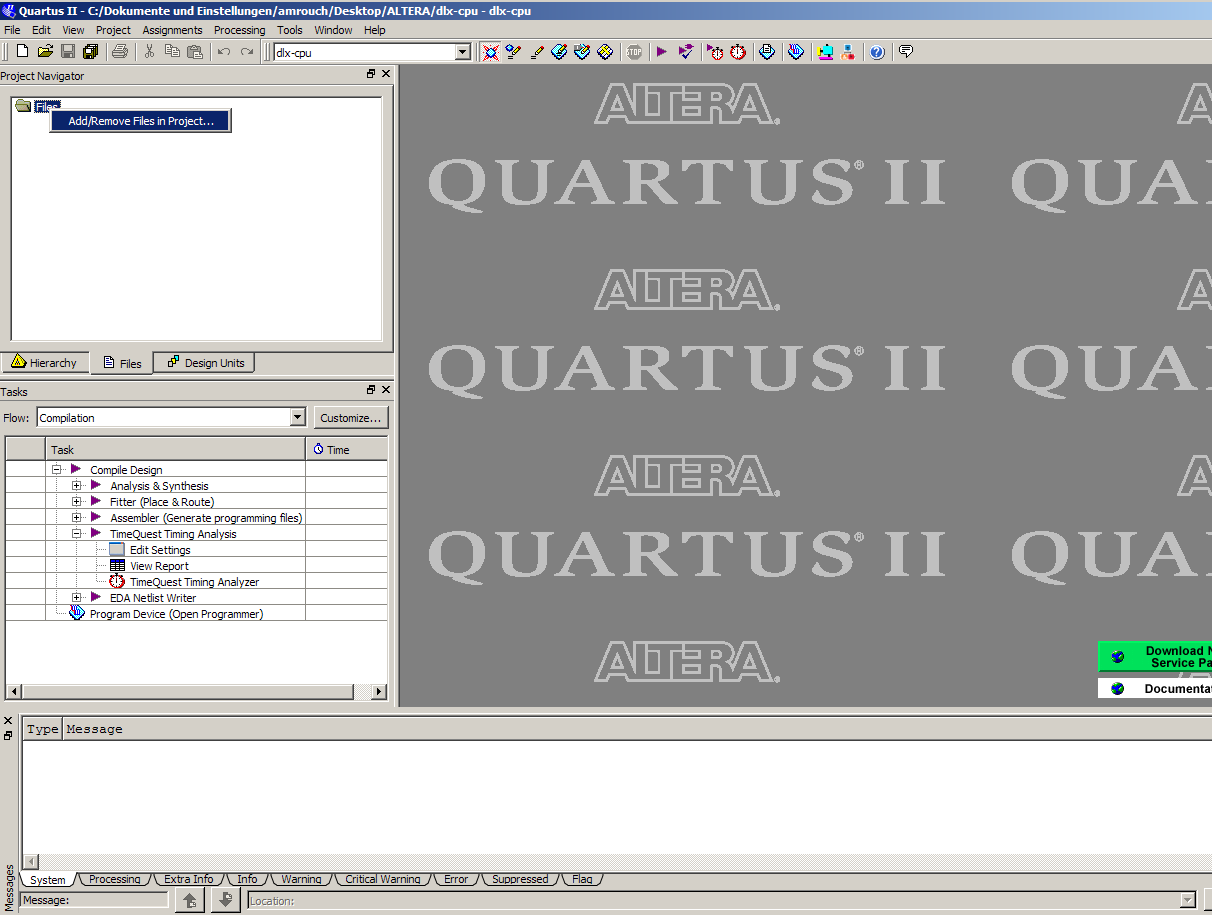


Figure 6‑3. Adding Files to The Project

To synthesize and to implement your project you have to choose “Start Compilation” from the Processing menu. This will finally create the bitstream file (.sof) that can then be initialized with your CPU instruction- and data-memory (IM.mif and DM.mif) and afterwards be uploaded to the FPGA prototyping board.

While synthesizing and implementing the design, many warnings will be printed. These warnings (unless created by a user modification, e.g. in the CPU) can be ignored. However, it should be mentioned, that it is very helpful to understand the meaning of these warnings when you are looking for a reason why something is working unexpected in hardware. The challenge here is, that the CPU and the IP cores create plenty of warnings, thus it is hard to locate the serious warnings.

## Initializing the software and uploading to the prototyping board

After you have finished the synthesizing and implementation step, you receive a bitstream of your Quartus project (.sof) that includes your CPU connected to an internal memory inside the FPGA. Now you have to initialize this FPGA internal memory with your application instruction- and data memory to execute your program on your CPU.

To initialize the bitstream with your application, you need your application as IM.mif and DM.mif files. However, compared to simulation with dlxsim or ModelSim you have to consider, that you have a limited amount of memory on the FPGA and therefore you have to adjust the position were the stack starts. For the usual simulation, the stack can start at address 0xFFFFC and is growing downwards. For hardware execution, this address is too big. For the current hardware prototype, you should use 0xEFFC. You can adjust the place where the stack shall start in the file “mkimgSettings”. This file has to be located in the directory of your application and it is evaluated every time you call a version of mkimg or mkall. If this file does not exist, some default values are used for all settings. You can find a version of this file in ~asip00/­ASIPMeisterProjects/­TEMPLATE\_PROJECT/­Applications/­TestPrint/.

The file “mkimgSettings” contains three parameters:

* STACK\_START: Here you can configure the address of the stack start. To work in hardware this value depends of the size of the available memory in hardware. For our current prototype, “0xEFFC” is the correct value.
* ADD\_NOPS: With this parameter, you can add additional NOPs between the real instructions of the application. For running in hardware, you need a value of 6 here, as our current CPU has some internal problem in some specific cases (due to some problems with the automatically created VHDL code of the CPU), which do not appear in ModelSim simulation! Note that a less number of number is still possible but we use 6 NOPs just for avoid any unexpected processor crashing. Please try to use a less number of NOPs like (4 NOPs) and see if the processor will still work correctly or not.
* MAX\_ALLOWED\_SUCCESSIVE\_NOPS: With this parameter, you can restrict the number of successive NOPs. Sometimes there are much more than three successive NOPs issued to the code, although three is the absolute maximum that the current pipeline structure needs for resolving pipeline dependencies, as explained in Chapter . For hardware, you have to use a value of 6 (due to some problems with the automatically created VHDL code of the CPU) and for dlxsim/ModelSim execution a value of 3 is sufficient. A value of 0 means that all NOPs are removed, except one NOP right after a Jump/Branch instruction (to fill the delay slot). This mode mimics the behavior that the CPU offers a data-forwarding unit that resolves all data dependencies. A value smaller than 0 (e.g. -1) means that the corresponding script will not be called at all and thus no NOPs are removed by the script.

The “mkimgSettings” contains a switch HARDWARE\_PROFILE to easily change between the settings for FPGA prototype and dlxsim. After you have adjusted the above parameters in the file “mkimgSettings”, you have to recompile your application to let the changes take effect on the created TestData files.

After the DM.mif and IM.mif-files for your application are created (see above and Chapter ) and the bitstream of your Quartus project is created (see Chapter  and Chapter ) you can initialize the bitstream with your application data by the two following steps:

* Processing ----> Update Memory Initialization File
* Processing ----> Start ----> Start Assembler

The bit file is the final bitstream of your specific CPU, of the surrounding framework, and with your application initialized to the memory. You have to configure the Quartus project that you want to use in the “env\_settings” as “Quartus\_NAME”. The bitstream in this directory will be used to create the application-initialized bitstream.

## Turning-On the FPGA Board and uploading the Bitstream

After you have created the final bitstream with the initialized memory, you can upload this bitstream to the FPGA Board. At first, you have to turn the FPGA Board on.

After the FPGA Board is running, you have to connect the FPGA to the PC and then initialize it with your Bitstream. Therefore, you are ready to upload your bitstream on the FPGA by clicking on “Programmer”. You can select which bitstream (.sof) you want to upload by clicking on “Add File”. You start the uploading by clicking on “Start”.

## Getting accurate reports for area, delay, and critical path

The results for area and speed for your synthesis result like created with the tutorial in Chapter  are not accurate. This is, because the provided framework contains main additions like a state machine to communicate with the LCD via the I2C bus or the data- and instruction memory and its connection to the CPU. These additions, which are needed for running the CPU on the hardware prototype, have a big impact on the measured size and speed of your CPU. Therefore, if you try to compare two different CPUs with the provided framework, you will mainly compare the framework with itself and it is hard to separate, which change in e.g. the CPU frequency is due to a change in the CPU or a more efficient optimization of the synthesis program due to a better interaction of CPU and framework.

To come around the above-mentioned problems, one might suggest synthesizing the plain CPU without any kind of framework to get accurate data without any impact of other components. Nevertheless, when you look at the output of this synthesis, you will notice, that all internal connections of the CPU will be automatically mapped to I/O-Pins of the FPGA, which has an impact of the size and speed of the synthesis result as well. This impact is due to the fact, that the I/O pins are rather slow compared to the FPGA-internal computation. However, you cannot force the synthesis tools to let the CPU connections unconnected, because then the synthesis tool would notice that there is no input to the CPU and that the output of the CPU is not used at all and thus it would remove the whole CPU for optimization reasons.

# Getting Area and Delay report

To accurately measure only your CPU we designed a new framework, which consists of the required files to create the Data/Instruction memories, dlx\_benchmark.vhd (toplevel entity for the whole project), the pin location file and the constraint file (.sdc), which can be found in the directory ~asip00/­ASIPMeisterProjects/­TEMPLATE\_PROJECT/­Quartus\_­Benchmark/. By using this framework, all the CPU connections will be mapped to the FPGA-internal BRAM memory and this will decrease the area needed to implement the project and will give more accuracy to compute the processor area and speed. To obtain the area and speed results for your CPU, your CPU files, and this special framework have to be synthesized and implemented.

**Area Report:**

In the Process sub window, expand “Fitter (Place & Route)” process. Double click on “View Report” will open new window containing the results needed. Try to find the number of Total logic elements as shown in Figure 6‑4.

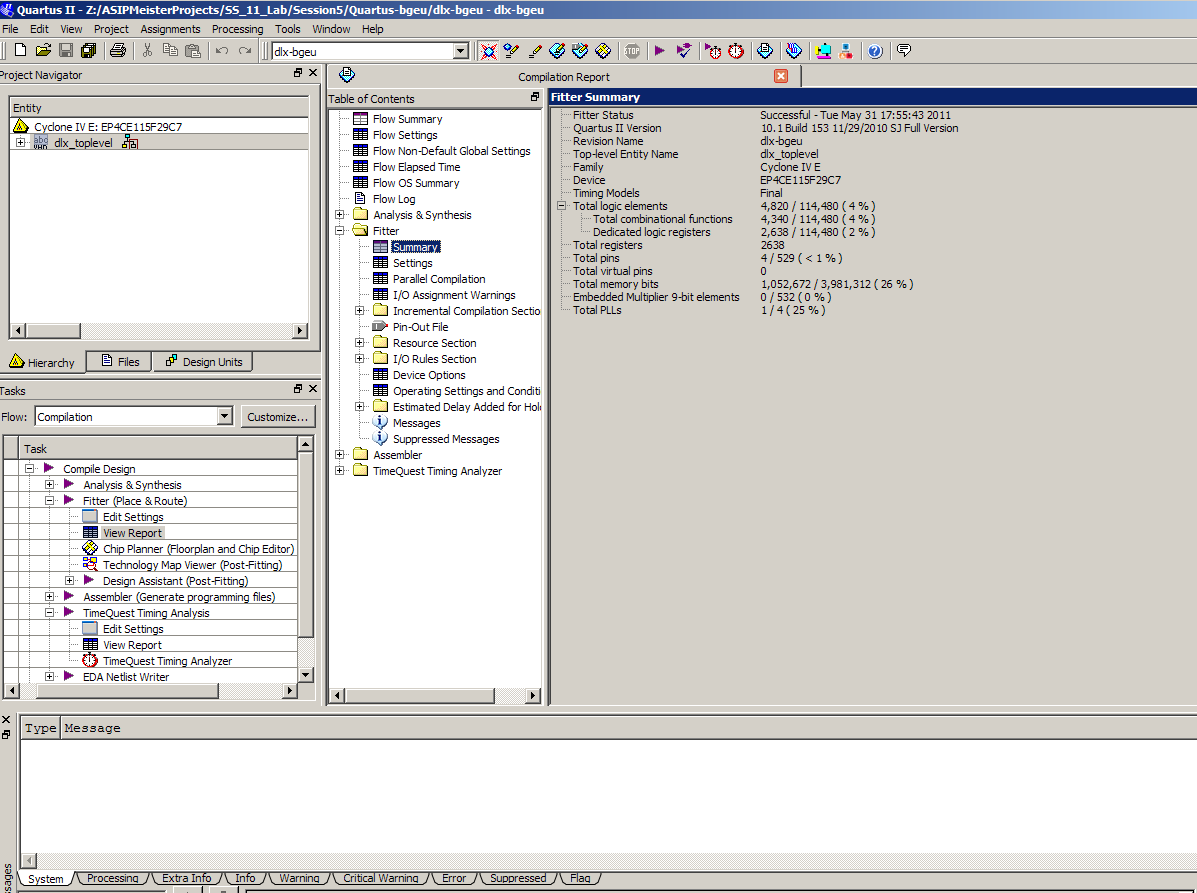


Figure 6‑4. Area Report

**Delay Report:**

In the Process sub window, expand “TimeQuest Timing Analysis” process, double click on “View Report” will open new window containing the results needed. Try to Fmax Summary as it can been seen in Figure 6‑5

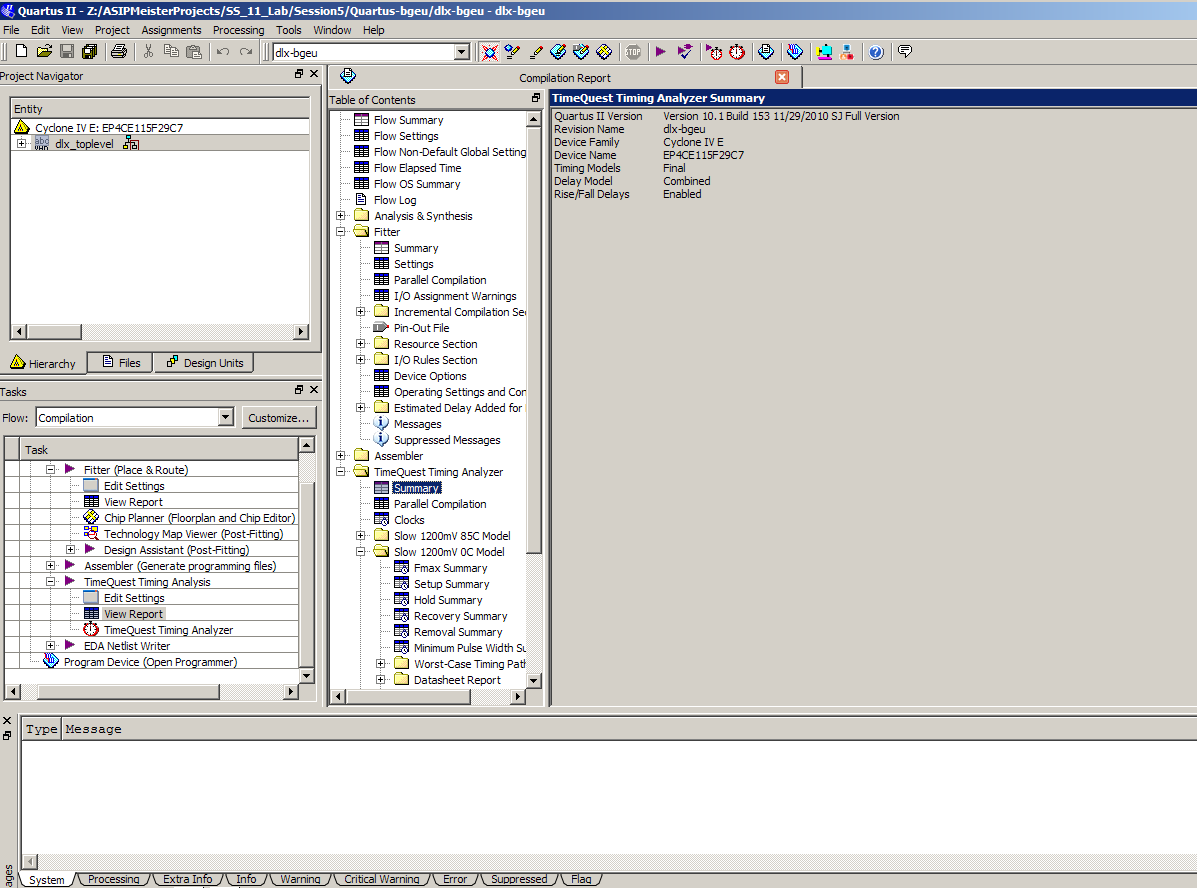


Figure 6‑5. Delay Report

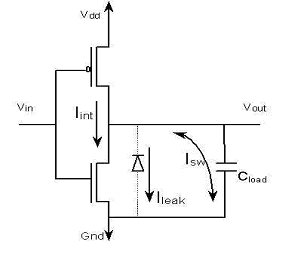
# Power estimation

In this tutorial we will learn how to estimate the power by using ModelSim to generate the switching activity of the design, and PowerPlay Power Analyzer Tool (from Quartus) to analyze the results and generate the final power report.

## Different types of power

Typically, power dissipation in a cell is subdivided into two different groups, dynamic power, and static power. **Dynamic power** is the power dissipated in a cell when the input voltage is actively transitioning. Dynamic power is further subdivided into two components switching power and internal power. **Switching power** is the power required to charge the capacitive load on the output pins of the cell. Switching power is shown in as the Isw switching current charging up the Cload capacitor. This component of power is calculated using the familiar ½ CV2 equation.

For switching power, all we need to know is Vdd and the capacitive load that is driven by the cell. Therefore, the library does not need to be characterized for this component of power dissipation.

Figure ‑   
Switching Power dissipation

**Internal power** consists of short-circuit power and power dissipated by charging the capacitive loads that are internal to the cell (not shown in the above circuit). Short-circuit power is the power that is dissipated due to the short period that paths in the cell are essentially short circuits. In the circuit shown above, Iint, that is the current path when the device is short-circuited, shows internal power. Every time the input Vin toggles, there is a short period of time where both transistors are turned on and there is a path from Vdd to ground. The longer both transistors are active, the higher the power dissipation. The circuit above is quite simple (an inverter) and there is only one path from Vdd to ground. For complex circuits, you could have dozens of potential short circuit paths. Internal power is dependent on the transition time of the input voltage Vin and the output capacitive load. These factors determine how long the short circuit is active. There is no easy formula to determine the power dissipated due to internal power, so the cell must be characterized for it.

The final component of power analysis is **leakage power**. In the circuit diagram in it is shown as leak current. This is the power dissipated when the circuit is in a steady state and it is due to the following factors inherent in transistors: reverse bias leakage current, sub-threshold current, or other second-order leakage power. Leakage power has become a major factor in power analysis.

At 180 nm and above, leakage power was typically less than 1% of the total power dissipation in a circuit. With 130 nm and below, the leakage power becomes a much larger factor, up to 50% of dissipated power in some cases. Leakage power should be characterized for each cell in the library.

To report the switching power, we need the switching activities of the design. The switching activity contains information about the static probability and toggle rate. The static probability can be calculated during a simulation by comparing the time a signal is at a certain logic state (state 0 or state 1) to the total time of simulation. The toggle rate is the number of transitions between logic-0 and logic-1 (or vice versa) of a design object per unit of time.

The previous power definitions can be concluded by these two equations:

Total Power = Dynamic power + Leakage power

Dynamic Power = Switching power + Internal power

## Estimating the power consumption

To estimate the power consumption for a specific application on a specific CPU you need to generate the switching activities, which are saved as a “value change dump” file (.vcd). This file is generated when the project is simulated (the application is executed on the CPU) using ModelSim. Then the .vcd file will be used as input to the PowerPlay Power Analyzer tool, which finally will generate the Power report.

### **Generate the “value change dump” file using ModelSim**

To generate the .vcd file, start ModelSim and create a project with the following files (you may also reuse an existing project if it contains the same files):

* All CPU VHDL files (from .syn folder of ASIP Meister)
* The application files (TestData.IM and TestData.DM)
* Testbench VHDL file (from the typical ModelSim simulation)
  1. Configure the CPU Frequency for which you want to run the power estimation. Open the ModelSim testbench (tb\_ASIPmeister.vhd), search for CLK\_HALF\_PERIOD, and change the value accordingly. Take care: For simulation-reasons this value corresponds to the time (in nano seconds) of a half clock period. For example, 10 ns half period correspond to 20 ns clock period, which is 50 MHz frequency.
  2. Compile the project Compile 🡪 Compile Order 🡪 Auto Generate
  3. Start the simulation: VSIM > **vsim -t 1ns work.cfg**
  4. Store the activities during the program execution in the test.vcd file by typing:  
     VSIM > **vcd file test.vcd**
  5. Add all signals of the CPU instance: VSIM > **vcd add -r test/dut/\***

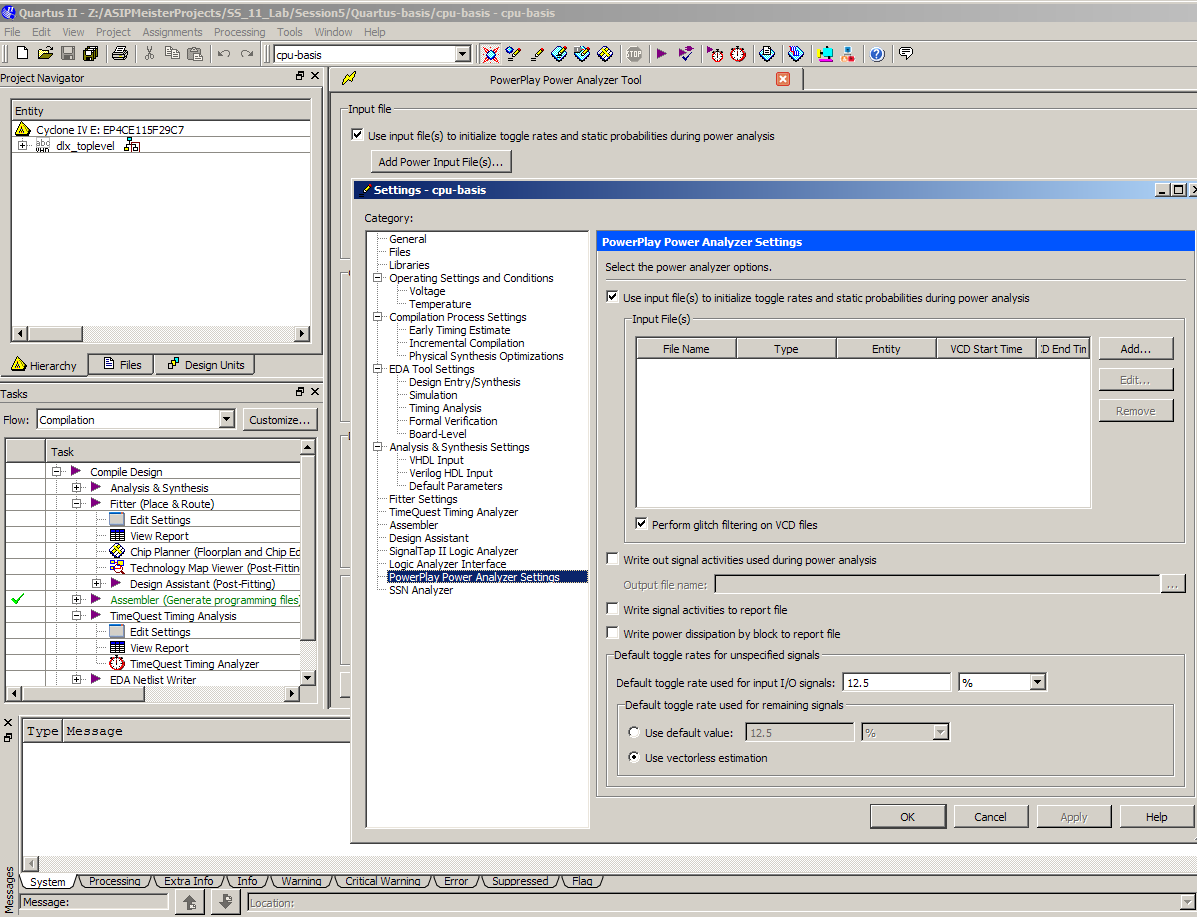
The entity name of the tutorial example testbench is *test* and the instance name of the device under test is *dut*. Using the -r switch with ModelSim’s ‘vcd add’ command will result in a large but significantly more accurate VCD file. VCD files can grow quite large for larger designs or even for smaller designs if the simulation run time is long.

* 1. Run the simulation: VSIM > **run -all**

The activities will be saved as test.vcd file.

### **Generating the power report using PowerPlay Power Analyzer Tool**

The second step is to create an Quartus project that you want to analyze. Make a new project and add only the CPU files from the ASIP Meister .syn directory to it. After you complier you project, you can start analyzing the power by clicking on PowerPlay Power Analyzing Tool from the “Processing” menu. Afterwards load the .vcd file generated in the previous step using “Add Power Input File(s)” as shown in



Adding the .vcd file to the PowerPlay Power Analyzer Tool.