Pre-Processor Asic Design Guide

Version 1.1

R. Achenbach, D. Husmann, M. Keller,
K. Mahboubi, C. Schumacher

November 29, 2002

Contents

List of Figures

List of Tables

Chapter 1

Overview

To allow the development of ASICs by different designers at the same time, it is necessary to agree about some conventions, which make it possible to build up a design from pieces done by different people. This includes the aspect of design re-use, where design blocks origined from other projects are imported. The use of appropriate conventions minimizes the effort to fit these blocks into the own design. And a third aspect is the introduction of new designers, which can profit from the experiences on which these conventions are based.

This document outlines the guidelines and conventions used by the ATLAS group of the Heidelberg ASIC-Lab. They are aimed especially at the design of the Pre-Processor Asic (PPrAsic), which is a pure digital ASIC described with Verilog.

In addition the tools and procedures used for the PPrAsic design are described. This includes design entry, synthesis, layout and simulation. In the following sections the PPrAsic and the design flow used for its development are introduced.

1.1 The Pre-Processor Asic

The Pre-Processor Asic (PPrAsic) is a digital ASIC used by the ATLAS Level-1 Calorimter Trigger. It has to prepare data received from calorimeters for subsequent application of algorithms by special trigger processors. It also has to provide facilites for readout of this data.

The PPrAsic as part of the ATLAS Level-1 Calorimeter Pre-Processor system is described in [10]. More details can be found in the specification [9] and the user and reference manual [1] of the PPrAsic.

The PPrAsic processes four channels of 10 bit ADC input data. As output three 10 bit words are provided, which are transmitted to two different trigger processors. In order to control and readout the PPrAsic two identical serial interface exist, each operating on two pre-processing channels. For more convenient simulation there also exists a two-channel version of the Asic, but only the four-channel version will be manufactured in silicon.

1.2 Design flow

The PPrAsic design is done using the hardware description language (HDL) Verilog for design entry. It is synthesized to a schematic netlist by the synthesizer from *Synopsys*. The resulting netlist is imported into the CAD software from *Cadence*. Its place and route tool *Silicon Ensemble* is used for the layout. Timing information is backannotated after synthesis and layout to the Verilog description. The Cadence tool *Pearl* is used to generate the required files and to perform static timing analysis.

The organisation of design data undelying the design flow is described in chapter 2 and the design entry with Verilog including coding conventions is documented in chapter 3. Synthesis is described in chapter 4, layout in chapter 5 and timing analysis in chapter 7.

On different levels of the design flow simulations with no, partial or complete timing information are done. The results are compared and have to be assured to be correct and consistent. The two Cadence simulators *Verilog-XL* and *NC-Verilog* are used. Chapter 6 describes the Verilog simulation environment in detail.

To make sure that the implementation of the PPrAsic meets the specification an independent simulation of the main processing part of the PPrAsic is used. This part, which performs the so-called *Bunch-Crossing Identification* (BCID) is modelled within a *Ptolemy* simulation environment. For more information see 6.7.

Other important aspects of the design flow are documentation and test. See chapters 8 and 9 for details.

And finally chapter 10 holds information about reviews which take place as an additional measure to ensure the correctness of the way the design is done and its results.

Chapter 2

Organisation of Design Data

Most of the design data is stored in a *CVS repository*. This includes the Verilog source code of the PPrAsic, scripts used for simulation and synthesis and configuration files. Some other files like the netlists or figures are stored in a global directory and there are also some files local to the user, which are related to the PPrAsic design data. This section describes the organisation of this files.

2.1 CVS Repository

2.1.1 Usage

For configuration management the *Concurrent versions system* (CVS) is used. This system holds the code in a central database called *repository*. If someone wants to edit the code he checks out a local copy to his disk. After editing, he commits changes back to the repository. This mechanism makes it possible for several developers to work on the same code base. CVS takes care of merging files and resolving conflicts arising from different people editing the same piece of code. CVS also keeps track of older versions of source files. If changes are comitted, no data are deleted. The differences to the previous version of the file are stored. This allows to access the complete history of the source.

To use CVS you have to set up the environment variable `CVSROOT` with the location of the repository you want to use. For the PPrAsic this should be `/cad11/atlas/cvs_open`. If you plan to write data back to the repository you have to be member of the UNIX group `atlas`. To checkout the current version from the repository use the command `cvs checkout <modulename>`. For the PPrAsic the modulename is `pprasic`. CVS than creates the file hierarchy in your current working directory. To update the files in a local copy of the source tree with the current versions from the repository use `cvs update` in the directory you want to update. To commit your changes back to the repository use `cvs commit`. For more information look at the cvs man page (`man cvs`).

2.1.2 PPrAsic Module Content

The CVS module `pprasic` contains several directories related to the different steps of the design flow and some files in the top directory:

- `verilog`: Verilog source code directory. Here the actual PPrAsic design resides.
- `syno`: Synthesis directory, contains all scripts required to generate a netlist from the Verilog source code.

- `netlist`: This directory contains the scripts required for simulation of the synthesized netlist. Synthesis results are written here.
- `se`: *Silicon Ensemble* files required for layout.
- `stimulus`: Stimulus vectors and files for simulations.
- `doc`: PPrAsic documentation.
- `CHECKLIST`: Checklist for simulations required for a complete test of the design.
- `TESTSDONE`: Results and log of the tests performed. Based on the list in `CHECKLIST`.

There exists a web interface [5], where you can view the content of the repository, compare different versions or look at specific revisions of files.

2.2 Global Files

The directory `/cad11/atlas/cad/pprasic` contains certain global files, which are not managed by CVS. This means that you have to communicate with the other developers before you edit this files in order to avoid conflicting changes.

The following subdirectories are available:

- `netlist`: Netlists of the PPrAsic corresponding to certain versions of the Verilog source. The naming scheme of the files and CVS tags used to relate the files with source code versions are described in the file `README` in this directory.
- `pspice`: Calorimeter pulse data files generated from the PSPICE simulation of the calorimeter electronics.
- `figures`: Figures used in the LaTeX documentation (User Manual and Designguide). The files with the suffix `.fig` are generated with XFig. The corresponding eps-files are used by LaTeX.
- `db`: Synopsys database files corresponding to the netlists in the `netlist` directory. Version numbers are the same as used for the netlists. These database files contain the schematics of the synthesized circuit.
- `simlogs`: Log-Files of simulation runs. Files with a name of the form `index.modulename.log` contain lists of simulations done with the module `modulename` as top module. The index files contains references to detailed log files also located in this directory.

2.3 Local Files

There are some files in the home directory of the designer which contain information relevant to the design flow.

The file `.synopsys_dc.setup` contains Synopsys commands, which are not specific to a certain design. An example of this file could be:

```
company  = "ASIC-Lab Heidelberg"
designer = "Name of Designer"
```

There are some environment variables, which have to be set for the design flow to work correctly. This is done in the file `.bashrc` in the home directory of the designer. The following code has to be inserted there:

```
source /usr/local/cad/scripts/cds4.42rc
source /usr/local/cad/scripts/synorc

export AMS_DIR=/cad/libs/ams3.12
export CDSDIR=/cad/products/cds4.42

export CVSROOT=/cad11/atlas/cvs_open
```

Chapter 3

Design Entry

3.1 Coding Conventions

The data constituting the ASIC design consists of a large amount of Verilog code, including test benches and associated modules. In addition there are some scripts for setting up and processing the synthesis and some parameters needed for layout and the rest of the design flow. From all this informations it has to be possible to create the final chip by using the CAD tools. All information necessary for doing this should be stored in a systematic and consistent way.

Especially for the Verilog code there have to exist some conventions about the coding style, the naming of modules and signals and the way to implement functionality in an actual circuit.

In this section the conventions are described which were adopted for the design of the PPrAsic. They focus mainly on the Verilog coding which is by far the biggest contributor to the whole design.

3.1.1 Code organisation

The Verilog code is stored in an own directory, synthesis and layout data is stored in others. The following guidelines should be followed:

1. Each Verilog module gets its own file which should have the same name as the module plus the suffix "`.v`". For guidelines refering to the naming of modules see section 3.2.

2. Numerial constants should be defined symbolically by a `` `define `` statement. This helps to reduce redundancy. If you change one parameter of your design you should't have to change your code at more than one location.

3. All `` `define `` statements should be stored in include files. The simulator remembers defines across modules. If the same symbol is defined with two different values, this can lead to strange errors. The central definition in one file avoids these inconsistencies. The names of include files should start with "`Inc`".

4. Use `parameter` statements for modules where you need the same module with the same logic but only different parameters like bus widths etc. This also helps to minimize redundancy.

5. Write portable code. Encapsulate technology specific code in more general modules. Minimize the number of modules and the amount of code you have to change if you change technology. Use the synthesizer to map to specific technology wherever possible.

```
// $Id: entry.tex,v 1.5 1999/12/08 16:20:48 huebner Exp $
//
// Example Verilog module demonstrating coding conventions
//

module MyModule (BidirectionalSignal,
                 AnotherOutput,OtherOutput,Output,OutputWithVeryLooooooongName,
                   YouKnowItItIsAnOutput,
                 Input1,Input2,Input3);

  inout BidirectionalSignal;  // bidirectional port

  output AnotherOutput,OtherOutput;  // some outputs
  (...)
  input Input3;  // input signal

  always @(Input1 or Input2 or Input3)
    begin
      Output = Input2;
      if (Input1 == 1)
        Output = Input3;
    end

endmodule
```

Figure 3.1: Example for Verilog module conforming to the coding conventions

3.1.2 Code formatting

A consistent formatting style should be used for the Verilog code. This allows other people than the designer to understand the code without being distracted by formal differences. An example (Figure 3.1) illustrates the following rules:

1. Each file should begin with a comment block stating the purpose of the module. The first line of the file should include the CVS Id tag. For a new file this should be `$Id$`. See section 2.1.1 for more information about CVS.

2. The I/O signals contained in the module definition should be ordered in the following way: All `inout` signal come first, than all `output` signals, than all `input signals`. Within each category the signals should be ordered alphabetically. Each category gets its own line. The categories should be aligned to the same column. If one line is not enough for all signals of one type, then the list should be continued in the next line indented by two spaces.

3. The module definition should start at the leftmost column of the file. The body of the module is indented by two spaces. The body of each `begin`, `initial`, `always`, `task` or `function block` is indented by a further two spaces. The same is valid for the statements executed within `if`, `else`, `while`, `case` or `for` constructs. They should start on a new line and be indented by two spaces in respect to the `if`, ... statement.

3.1.3 Circuit concepts

Despite the formal conventions making it easier to read modules and to interface to other modules there are conventions refering to the type of circuit that is described by the Verilog code. These

try to avoid some pitfalls, make it easier to test the design in a stringent way and try to ensure that the simulations correspond to the behaviour of the real chip as far as possible.

1. Make a synchronous design as far as possible. Many problems can be avoided if you ensure that you have a common clock with the same phase for all sequential elements. Use the positive edge of this clock to operate all flip-flops etc. in your entire design.

2. If you have to use an asynchronous element in your design, document the reason for using it and document the constraints it poses on the surrounding circuit. Encapsulate it in a synchronous module if possible.

3. Ensure that after power-up or reset the system is in a stable state. No activity should occur without changing the input signals.

3.2 Naming conventions

A consistent naming scheme makes the code easier to understand, avoids naming conflicts and improves code organisation. Several guidelines are given in the next sections.

3.2.1 General

1. Use english names. You never know who wants to look at your design. More probable you already know somebody who wants to and doesn't understand german.

2. Use meaningful names. Avoid uncommon acronyms. `BottleState` is better than `BtlSt`. The increase in comprehensibility is more valuable than the saving of characters.

3. Use names which make sense when used in context. `if (BottleClosed) openit;` is better than `if (BottleState == 0) openit;`.

4. Don't use special characters in your names. This includes the underscore "_". Some tools get confused by these characters.

5. Use uppercase letters to indicate the boundaries of words in names made up of several words.

3.2.2 Module names

1. The modules belonging to the actual design and specific to that design should start with a common prefix. This prefix should be short. This can violate the rule about meaningful names, but since this prefix is included in all module and file names a long prefix would be annoying. For the PPrAsic the prefix "`Pa`" is used.

2. Synthesizable modules should start with an uppercase letter. Top modules of test benches should start with "`test`" followed by the module they test. Additional modules used by the test bench and not belonging to the actual design should start with "`tb`" followed by a meaningful name starting with an uppercase letter.

3. General modules which are not specific to a particular design should start with a common prefix. We use "`Gen`". Examples are registers, counters, etc. In many cases such modules would be parametrized by one or more `parameter` statement.

4. Try to reflect the hierarchy of the design in the names of the modules. Modules which can only be used as instance in another module should have the name of the instantiating module followed by one or more words identifying the module. For example the readout module `PaReadout` of the PPrAsic could have the submodules `PaReadoutCtrl` and `PaReadoutMemory`. This convention can lead to very long names, so be concise and use short, meaningful names.

3.2.3 Signal names

1. Standard names of signals should start with an uppercase letter.
2. Standard signals should have standard names. Use `Clk` for the system clock, `Reset` for the global asynchronous reset signal.
3. Use consistent names through hierarchy. Signals having the same function in different modules should have the same name.
4. Signals using negative logic should end with "`Bar`".
5. Bus indices count down from the maximum value to the minimum value. Example: `reg [31:0] BusData`.

3.2.4 Instance names

1. Names of Verilog module instances should be short and contain only capital letters.
2. The top level instance of the circuit as instantiated in the test bench should have the name `TOP`.

3.3 Automatic register generation

Most of the registers, which store settings for the PPrAsic, are generated automatically from simple text configuration files. There are two register blocks, a global one and a trigger channel specific. These blocks are described by the files `PPrAsicGlobalReg.conf` and `PPrAsicChannelReg.conf` located in the `verilog` directory of the CVS module.

The format of the files is the same as used by the *Hardware Diagnostics, Monitoring and Control (HDMC)* Software [6], which is used to operate the Pre-Processor Test-System. This software will also be used to access the registers of the PPrAsic. For a description of the configuration file format see the HDMC documentation [7].

To generate the registers use the command `make reg` in the *verilog* subdirectory of the CVS module. This creates the two Verilog files `PaReg.v` and `PaChannelReg.v`, which provide modules to be instantiated in the PPrAsic code. Two include files `IncPaReg.v` and `IncPaChannelReg.v` are created, which contain definitions of the bit widths of register entries. These have to be included in the files, which use multi-bit outputs of the register modules. If you add new registers or add or delete register entries, you have to adjust the instantiations of the register module.

The `make reg` command also creates two Perl files `PaReg.pl` and `PaChannelReg.pl` which are used by the stimulus generation scripts (see section 6.2).

Chapter 4

Synthesis

Synthesis is done by the *Synopsys DesignCompiler*. It can be started interactively with a graphical user interface by the command `design_analyzer` or in batch mode using `dc_shell`. Prerequisite is the execution of `/usr/local/cad/scripts/synorc` in your `.bashrc`. Online documentation for *Synopsys* can be started with the command `sold`.

4.1 How to perform synthesis

To perform a synthesis, one should change to the `syno` directory of the PPrAsic CVS repository. Synthesis is started with the command `syn` with the module name as argument. For example the top module of the PPrAsic is synthesised with the command `syn PaBigTop`. Output could be redirected to a log file as: `syn PaBigTop > syno.out` .

The synthesis command uses the same module list as the simulation to determine, which Verilog files are required for the design. See section 6.3 for details about the module list.

There is an option file, which can be used to control synthesis. It is located in the `syno` directory and is called `syn.modulename.ls`, where modulename is the name of the module being synthesised. A valid line in this file could contain at the beginning of the line one of the following keywords:

- `SCAN`
- `JTAG`
- `PAD`

The keyword `SCAN` indicates that internal scan paths should be inserted during synthesis. The number of the required scan paths and the list of the flip-flops to be included in each scan path is defined in the file `modulename_scan_chains.lst`, modulename being the name of the module being synthesised. The `JTAG` keyword means that the Boundary Scan interface should be added to the design. And finally the keyword `PAD` would cause pads to be defined and inserted at the top level of the design.

4.2 Scripts used by synthesis

Upon execution of the `syn` shell script on the target module two perl scripts are executed to generate different dc_shell scripts to be included in design compiler. Both perl scripts get the module name as the argument. At the end of the `syn` shell script the actual synthesis is started in batch mode. The two generator perl scripts are:

`make_anascr:` based on the content of the module list file produces the dc_shell script `read_verilog.<modulename>.scr`, which contains commands to *analyze* and *elaborate* all related submodules.

`make_synscr:` based on the contents of the synthesis option file, i.e. `syn.<modulename>.ls`, generates the actual dc_shell script files to synthesise the module. The generated files are `optimize.scr` and `syn.<modulename>.scr`. The latter contains all the synthesis procedure which should be performed on the target module.

An example of a dc_shell script file generated by the `make_synscr PaBigTop` commad, together with the `syn.PaBigTop.ls` option file containing `SCAN`, `JTAG` and `PAD` keywords on three separate lines, is the following (content of the `syn.PaBigTop.scr` file):

```
/* dc_shell script: syn.PaBigTop.scr */

include scripts/read_mem.scr
include scripts/read_verilog.PaBigTop.scr
include scripts/optcmds.PaBigTop.scr
include scripts/scan_definition.scr
include scripts/jtag_insertion.scr
include scripts/pad_insertion.scr
include scripts/optimize.scr
include scripts/scan_insertion.scr
include scripts/hookup_scan_to_jtag.scr
include scripts/make_scan_visible.scr
include scripts/make_jtag_visible.scr
include scripts/write_design.scr
include scripts/write_netlist.scr
quit

/* end of script */
```

Here is a short description on each of these dc_shell scripts:

`read_mem.scr:` Here the AMS memory blocks (RAMs) are read into the design link library.

`read_verilog.PaBigTop.scr:` All required submodules are *analysed* and *elaborated* in this script.

`optcmds.PaBigTop.scr:` Contains the optimization constraints. At the moment the only constraints are the clock networks (clock period and waveform definition).

`scan_definition.scr:` Here, the required number of internal scan paths, which is set in the first line of the scan chain file `PaBigTop_scan_chains.lst`, is determined. This is reqired by the next script.

`jtag_insertion.scr:` The JTAG input/output ports/signals are defined in this script. First all functional logic is grouped into ***Core*** using the `group_into_core.scr` dc_shell script. Then the total number of internal scan chains is set as JTAG data registers and equipped with instruction numbers. Identification register for the chip is also defined here. Finally the actual jtag insertion is performed. At this stage the unmapped JTAG logic is fixed using the `fix_jtag.scr`, which in turn makes use of the `onezero.pl` perl script.

`pad_insertion.scr:` The JTAG logic plus the ***Core*** logic is first grouped into ***TOP*** using the `group_into_top.scr` dc_shell script. Then pads are defined and inserted on the top level of the design.

`optimize.scr:` Here the design is uniquified, linked and finally compiled. If internal scan has been defined the `-scan` compile option is used to get a test ready design.

`scan_insertion.scr:` The internal scan chains, based on the contents of the `PaBigTop_scan_chains.lst` file, are inserted and routed in ***Core*** design. THe content of the scan chain file is read using a perl script `read_scan_chains.pl`, getting as argument the file name containing the flip-flop list for each chain and the chain number.

`hookup_scan_to_jtag.scr:` The internal scan chains are manullay connected to the JTAG logic. A final reoptimization is also performed here, which doesn't modify the ***Core***, JTAG logic and PADs (because of an existing dont_touch attribute on these objects at this stage). This is to optimize the added combinatorial logic.

`make_scan_visible.scr:` Information about internal scan is refreshed.

`make_jtag_visible.scr:` Information about boundary scan is refreshed.

`write_design.scr:` Design is written out in database format (PaBigTop.db). Timing and area reports are also generated and written to `PaBigTop_timing.rpt` and `PaBigTop_area.rpt` .

`write_netlist.scr:` The netlist file in verilog format is produced.

At the end of `syn` shell script, on return from the dc_shell, the netlist is modified for PAD names to get rid of name clashes and to simplify the layout process. This is done by running the perl script `modify_padnames`, which takes as argument the name of the target netlist file. The final output has the name <`modulename`>`.v` .

Chapter 5

Layout

The layout of the PPrAsic is generated in *SiliconEnsemble 5.2* (*SE*). Final checks and manual layout are done in the *Cadence Design Framework*.
SE is controlled via several files which can be found in the global repository.
Before startiging the layout, add the spare cells manually in the netlist `PaBigTop.v` generated with *Synopsys*. Add the module instantiation
(`spareCells SPARE ( ) ;`) directly after the output declaration in module `PaBigTop`.

5.1 Generation in Silicon Ensemble

To generate the layout execute the following files in *SE*

- `readDesign.mac`: The macro file is used to set up the environment and to read in the design data base. It reads all LEF-, timing-, Verilog-, SDF- and DEF- files.
  - **LEF**-files contain the layout information for the technology, core cells and periphery cells:
    - `cup.lef` Original files from AMS,
    - `HRDLIB_3M.lef` Original files from AMS,
    - `IOLIB_3M.lef` Original files from AMS,

    for the memory blocks:
    - `dpram128x11.lef` Optimised AMS files,
    - `spram1024x8.lef` Optimised AMS files,
    - `spram256x11.lef` Optimised AMS files,

    and for the temperature sensor:
    - `tempsens.lef`.
  - `mycup3.3Vt.gcf` reads the **timing** information of the used libraries:
    - `AMS_DIR/artist/HK_0.6/HRDLIB/timing-3.3V.ctlf`,
    - `AMS_DIR/artist/HK_0.6/IOLIB_3M/timing-3.3V.ctlf`,
    - `ATLAS_DIR/ams/mem_new/spram1024x8/cadence/spram1024x8_lib/timing-3.3V.ctlf`,
    - `ATLAS_DIR/ams/mem_new/dpram128x11/cadence/dpram128x11_lib/timing-3.3V.ctlf`,
    - `ATLAS_DIR/ams/mem_new/spram256x11/cadence/spram256x11_lib/timing-3.3V.ctlf`.

    The operating conditions can be adjusted in this file as well.

- **Verilog**-files contain the logical description for the fuctional design, the spare cells and the temperature sensor:
  - `PaBigTop.v`,
  - `spareCells.v`,
  - `tempsens.v`,

  and for the memory blocks:
  - `dpram128x11.v` Original files from AMS,
  - `spram1024x8.v` Original files from AMS,
  - `spram256x11.v` Original files from AMS.
- **SDF**-file
  - `constraints.sdf`

  contains the timing constrains generated by *Synopsys*.
- **DEF**-file
  - `DEF/power_corner_new.def`

  contains the locations of the corner cells, power pads and the temperature sensor. The pad rings is defined in this file as well.

The design is stored in the database `DB/loaded`.

- `plan.mac` is used to create the floorplan.
  - The I/O placement file `pprAsicIO.ioc` is read in for pad placement.
  - `movemem.mac` is executed to place the memory blocks.
    All RAM blocks are placed relative to the corner cells of the chip. The memory blocks for each channel are grouped together.
  - `fillperi.mac` is executed to connect the pad ring. Also power routing is done within this file.

  The design is stored in the database `DB/powerRouted`.
- `qplace.mac` is used to place all standard cells. The design is stored in the database `DB/qplaced`.
- `ctgen.mac` does the DEF file exchange between *SE* and *Envisia Clock Tree Generation* (*CT-Gen*). This macro file calls:
  - `ctgen.cmd`. This comand file for (*CT-Gen*) uses:
    - `CTGEN/ctgen.const`.
      This constraints file defines the desired skews and delays. Also the switched clock branches are be described here.

  The design is stored into the database `DB/clkPlaced`.
- `wroute.mac`: With this file the final routing of the chip is done. The design is stored into the database `DB/wrouted`. If `wroute.mac` ends with routing errors, start *WarpRoute* with different parameters, i.e. run `wroute_cl.mac`.
- `output.mac` is used to write out logical and physical `.sdf`, the post layout `.v` and `.def` and a `.rpsf` file. It also executes `fillcore.mac` to place the feed trough cells. The design is stored in the database `DB/final`.

After post layout simulation, described in Sections 6 and 7, it may be necessary to modify some routing manualy (e.g. if one bus line is slower than the rest).
This can be done by deleting the bad net manually and rerun *WarpRoute*. (the command sequence for *WarpRoute* is given in the comment at the end of `wroute_cl.mac`).
Since the space between the power pads and the power ring is limited, *SE* does not connect some of the pad terminals to the ring. Due to the editing inflexibility of *SE*, these errors should be corrected in *Virtuoso*. Most of the grid and antenna errors should be cross-checked with *Diva*.

5.2 Verification in Cadence Design Framework

After the script driven generation of the layout in *SE*, the verification is done in the *Cadence Design Framework*.

5.2.1 Import of SiliconEnsemble Output

- **Import Verilog**

  Import the post layout verilog code `VERILOG_layout.v` in *ICFB*:
  File ⇒ Import ⇒ Verilog ...
  Load in the parameters from `vin_parameter.sav` and adjust them according to your library structure.

  The relevant libraries and options are:

  - Reference Libraries
    The following files has to be included:
    * `$CDSDIR/tools/dfII/etc/cdslib/basic`,
    * `$AMS_DIR/artist/HK_0.6/IOLIB_3M`,
    * `$AMS_DIR/artist/HK_0.6/HRDLIB`,
    * `$ATLAS_DIR/ams/mem_new/dpram128x11/cadence/dpram128x11_lib`,
    * `$ATLAS_DIR/ams/mem_new/spram256x11/cadence/spram256x11_lib`,
    * `$ATLAS_DIR/ams/mem_new/spram1024x8/cadence/spram1024x8_lib`,
    * `PPRASICTEMPSENS`,
  - -V Options
    Fill in the stubs files. Stubs files, consist only of the port declaration of the verilog description.
  - Schematic Generation Options ⇒
    Through CellView To Be Used For Port Shorts
    Select the `patch` cell of the `basic` Library for the last assign statements in the verilog code.

  After verilog import, copy the schematic of the temperature sensor in the schematic cellview of the top cell `PaBigTop`.

- **Import DEF**

  Before importing the DEF file, change the library path of `IOLIB_3M` and `HRDLIB` libraries.

  from `$AMS_DIR/artist/HK_0.6/HRDLIB` to `$AMS_DIR/artist/HK_0.6/LEF/SE/HRDLIB`
  from `$AMS_DIR/artist/HK_0.6/IOLIB_3M` to `$AMS_DIR/artist/HK_0.6/LEF/SE/IOLIB_3M`

Import the DEF file of the layout `PaBigTop.def` in ICFB:
FILE ⇒ IMPORT ⇒ DEF ...
The import Form is self explaining, use the same reference library names as in verilog in form.

After import change the library paths back to their original paths.

5.2.2 Manual Changes

- **Power pads**

  As explained in Section 5.1 *SE* does not connect some of the pad terminals to the rings. This is done manually.
  When opening the layout view from the library manager, *ICFB* will start the LAYOUT-XL tool. Stay in this tool while connecting the terminal A of the supply pads to the appropriate ring and correct all shorts in nets crossings this new supply connection.
  After that, change to the LAYOUT tool.

- **Terminals**

  For the later LVS check, add the terminals in the layout, just place it as a metalpinning layer over every pad and enter the proper net name in the query window.

- **Slots**

  According to the AMS 0.6μm CMOS design rules 4.5, 4.7 and 4.9 metal planes bigger than 20x300μm needs slots. This feature is not included in *SE*, therefore it has to be done manually. All errors should occur on the power ring, the pad rings and the power stripes. AMS states that slots has to be done on the pad rings, too. Realistically it is not applicable to put slots in the pad cells, but slotting the `PERI_SPACER` cells works without problems. Also the via arrays cannot be slotted.
  Keep in mind the current flow direction.

5.2.3 Verification

- **DRC**

  The Design-Rule-Check (DRC) is an interactive process. Examine the errors, possibly correct them and run the checker again , until verification shows that the remaining errors are uncritical.
  The DRC needs a **very large** (> 1GB) temporary directory. Check and export the `DRCTEMPDIR` if critical. This temporary directory will be used by all *Diva* tools: *DRC*, *Extract* and *LVS*.
  First run the DRC tool with the SE_NOTCH_CORRECT switch. This corrects all notch errors, which were produced during the routing process in *SE*
  After this, run it without switches.
  For verification purposes use ASIC-TOOLS ⇒ FIND DRC MARKERS. This tool allows to choose the type of error to view.
  Errors which can be ignored:

  - `DIFF intersects NTUB`
    This error emerges in pad cells. According to AMS, this is a problem of the checking software.
  - `MET3 coverage < 30%`
    Due to the large area consumption of the memory cells and the fact that the RAMs are layouted without MET3, it is not possible to achieve a coverage of 30%, however AMS gave their agreement in this case.

- `MET1` $>$ 20μm x 300μm `without slots`
  `MET2` $>$ 20μm x 300μm `without slots`
  `MET3` $>$ 20μm x 300μm `without slots`
  As mentioned above via arrays and the pad rings which cross pad cells will not be slotted, which results in this kind of DRC errors. Verify the errors in the layout by eye, and reduce the size of the error markers if possible.

All other errors must be eliminated. Most of them should be violations of the minimum spacing between metal shapes, which occur at the edges of the memory blocks.

- **LVS**

  Before running an Layout-Versus-Schematic (LVS) in macro mode the following have to be done.

  - **Views**:
    The temperature sensor cell has to be flattened with the ONE LEVEL option in:
    EDIT ⇒ HIERARCHY ⇒ FLATTEN
    All other cell `layout` views have to be changed to `abstract_mlvs` views. This should be done with the *Cell Ensemble* tool.
    TOOLS ⇒ FLOORPLAN/P&R ⇒ CELL ENSEMBLE
    Change the views with:
    FLOORPLAN ⇒ REPLACE VIEW.
  - **Extract**
    The extraction should be done from the shell, this process uses the DRCTEMPDIR extensively, especially in flat mode. The command is:
    `ivVerify extract -lib <l> <c> <v> -macro -join -rf divaEXT.rul -rl TECH_CUP`
    <l>= Library Name
    <c>= Cell Name
    <v>= View Name
  - **Macro LVS**
    Check if the macro LVS mode is set in the HIT-KIT UTILITIES menu of *ICFB*.

  Now open the LVS form. Select the run directory, the schematic and the extracted view. The RULES FILE has to be set to `divaLVS.rul`, tick the RULES LIBRARY and fill in `TECH_CUP`. From the options mark only TERMINALS.

  AMS told us that the macro LVS does not check the power nets and as tests have shown, the LVS will produce errors in the power nets. But this is not a "real" problem, because the chip will be checked flat after stream out.

5.2.4 Output

Before the stream out, the logos, butterflies and the scribe line are placed. Now check them manually!
Export the layout with the self-explaining stream out form of *ICFB*:
FILE ⇒ EXPORT ⇒ STREAM ...
If all tests are sucessfull, this GDS2 file will be sent to the vendor.

5.2.5 Output verification

From experience, the stream out process is not reliable, therfore a carefull check is requiered.

- **Stream in**
  Create a new library for the import, stream in the GDS2 file from *ICFB*:
  FILE ⇒ IMPORT ⇒ STREAM ...
  Do not specify the ASCII TECHNOLOGY FILE, since this is already done in the new stream in library.

- **Scribe line and logos**
  Open the layout and delete all logos, butterflies and the scribe line.

- **DRC**
  Run again a DRC (see Section 5.2.3) and verify that the errors found are the same as the DRC errors before stream out. Otherwise the the GDS2 File is possibly corupted

- **Extract**
  As contrary to Section 5.2.3, now the extraction is done in flat mode for verification of the power nets. The extraction on a *HP-9000/785* will take raughly 83 hours or on a *Sun Fire280* 45 hours.
  The command is:
  `ivVerify extract -lib <l> <c> <v> -join -rf divaEXT.rul -rl TECH_CUP`

- **flat LVS**
  Unset the macro LVS mode in the HIT-KIT UTILITIES menu of *ICFB*, the flat LVS should be started with the same options as the macro LVS.
  If some devices have size errors, try to write a correspondence file.

Chapter 6

Simulation

Simulations take place at various stages of the design flow. They all use common test benches and stimulus vectors. The results are compared to make sure that the simulations a consistent for all stages of the design. Test benches are described in section 6.1, generation of stimulus data for the serial interface of the PPrAsic in section 6.2.

The first stage is a functional simulation of the Verilog code without timing information. This is used to achieve logical correct behaviour of the design. How this simulation is done is described in section 6.5. At later stages the simulation is repeated with additional timing information of schematic and layout. The final simulation includes all delays of standard cells and nets with final placement and routing after layout. Timing simulations are described in section 6.6.

In order to check that the implementation of the BCID algorithm on the PPrAsic is correct and conforming to the specification, a parallel simulation of the algorithm with the *Ptolemy* [8] package is used. This is independent of the Verilog code but uses the same input data. The output of the Verilog and the Ptolemy simulation is compared to make sure that both simulations give the same result. The Ptolemy simulation and how to use it with the Verilog simulation is described in section 6.7.

6.1 Testbench

Simulation of Verilog modules requires a test bench, which provides the environment of the module, like input signals or models of surrounding circuits. It can also contain modules to analyse and check the output of the module under test.

The PPrAsic test bench (`testPaBigTop.v` for the 4-channel PPrAsic, `testPaTop.v` for the 2-channel version) contains modules providing ADC input (`tbDataSource`), the Level-1 trigger signal (`tbTrigger`), bunch-crossing demultiplexing (`tbPaBcDeMux`) and the cross-check of Verilog and Ptolemy simulation output (`tbPtolemyVerilogComp`). Other input data (`Clk`, `Reset`, etc.) are generated directly in the test bench module.

A special case are the signals of the serial interface of the PPrAsic. The module `tbSerialSink` reads the serial output signals of the Asic and prints the decoded data to the screen. The input data signals are provided by a Verilog file `ser.v`, which is included in the test bench. The file is located in the `stimulus` directory of the PPrAsic CVS module. It is generated automatically from stimulus input files, when the command `make serstim` is issued in the `verilog` directory or another command is used, which requires the serial stimulus data.

Name of Command	Description
`Raw <word>`	Send raw data
`Wait <cycles>`	Wait for a number of cycles
`Nop <cycles>`	Equivalent to `Wait`
`CmdLoadLut`	Send command to load LUT
`CmdStartReadback` <gaddr> <laddr> <mem>	Start Readback
`CmdReadoutReset`	Synchronous reset of readout
`Data <data>`	Send data word
`MemData <data>`	Send memory data word
`MemAddress <address>`	Send memory address
`CfgAddr <gaddr> <laddr> <mem>`	Send configuration address
`RegCfgAddr <gaddr> <laddr>`	Send address of register
`MemCfgAddr <gaddr> <laddr>`	Send address of memory
`LoadReg <file>`	Load registers from file
`LoadLut <file> <gaddr>`	Load LUT from file
`LoadPlayback <file> <gaddr>`	Load playback memory from file

Table 6.1: List of commands available in stimulus files for serial interface. `gaddrr` stands for global address, `laddr` for local address.

6.2 Stimulus Data for the Serial Interface

The files where you define the stimulus to the simulation are called `ser.username.stimlist` and `ser.username.stimlist`, where `username` has to be replaced by the account name of the user running the simulation, so each user has his own stimulus data. These files are located in the `stimulus` directory of the PPrAsic CVS module. They define a list of stimulus files, which contain the actual data fed to the simulation.

The format of these stimulus files allows to create sequences of serial data words by giving text commands, which are then converted automatically by a Perl script (`stimulus/make_serstim`). The format is described in section 6.2.1.

The serial interface is based on a synchronous protocol consisting of 13 bit words. Details can be found in [1]. These words can contain commands to the PPrAsic or user data to be written to memories or registers. Two flag bits are used to identify different types of words. The width of data payload then is 11 bit. Each word cycle of the serial interface consists of 13 cycles of the serial clock, which normally has the same frequency as the system clock.

6.2.1 Format of stimulus files for serial interface

Table 6.1 lists all available commands. They can be used as in the example below:

```
Wait 10
LoadReg init.reg
Wait 3
CmdLoadLut
Nop 20
#MemCfgAddr 1 1
#CmdStartReadback 1 1 0
```

This example waits for 10 serial word cycles, loads PPrAsic registers with values defined in the file `init.reg`, waits another 3 cycles, sends the command `LoadLut` to the PPrAsic and then generates 20 serial cycles of no operation for the serial interface. The symbol # is used to start a comment.

6.2.2 Commands of stimulus files for serial interface

`Raw <word>`

Send 13 bit raw data word to serial interface.

`Wait <cycles>`

Waits for the number of serial word cycles given with the argument. One cycle corresponds to 13 ticks of the serial clock.

`Nop <cycles>`

Equivalent to `Wait <cycles>`

`CmdLoadLut`

Send command, which loads LUT based on LUT register settings. See [1] for details.

`CmdStartReadback <globaladdress> <localaddress> <memoryselect>`

Send command, which starts readback of the block address by the arguments. See [1] for details of the addressing scheme.

`CmdReadoutReset`

Send command, which synchronously resets the event data readout. See [1] for details.

`Data <data>`

Send 11 bit data word.

`MemData <data>`

Send 11 bit data word to memory block.

`MemAddress <address>`

Send 10 bit address to memory block.

`CfgAddr <globaladdress> <localaddress> <memoryselect>`

Set configuration address according to arguments. Subsequent `Data`, `MemData` and `MemAddress` commands will go to the block selected by this address. For details of addressing scheme see [1].

`RegCfgAddr <globaladdress> <localaddress>`

Set configuration address to point to register denoted by argument.
This command is equivalent to `CfgAddr globaladdress localaddress 0`.

`MemCfgAddr <globaladdress> <localaddress>`

Set configuration address to point to memory denoted by argument.
This command is equivalent to `CfgAddr globaladdress localaddress 1`.

`LoadReg <file>`

Loads registers from a file. The register file format is described in section 6.2.3.

`LoadLut <file> <globaladdress>`

Loads LUT from a file. The file is parsed line for line and the resulting values are written in ascending order to the memory. The lines can contain decimal or hexadecimal values (hex numbers have to be prefixed by `0x`). In addition two commands are recognized: `Const <value> <count>` generates a sequence of constant values, `Ramp <startvalue> <count> <stepsize>` generates a value ramp. 1024 values are required to load the LUT completely.
The following example loads the LUT with a ramp with the values 5,6,6,6,6,7,7,7,7,...

```
Const 5 1
Ramp 6 1023 .25
```

The argument `globaladdress` has to be 1 to load the LUT of the first channel, 2 to load the second LUT and 3 to load both LUTs in parallel.

`LoadPlayback <file> <globaladdress>`

Loads LUT from a file. The format of the file is the same as for the `LoadLut` command. The only difference is that only 256 are required to load the playback memory completely.

`Verilog <command>`

Insert a user-defined verilog command into the serial stimulus files. This can for example be used to set the flag that enables comparison of verilog and ptolemy simulation results.

6.2.3 Register file format

The register file contains lines having the form

```
RegisterEntryName Value
```

where `RegisterNameEntry` is one of the entries that are defined in the register configuration files `PPrAsicChannelReg.conf` and `PPrAsicGlobalReg.conf` (see also section 3.3). The register entry gets loaded with `Value`.

One serial interface serves one bank of global registers and two banks of local registers. To define which bank of registers shall be loaded special modifier key words are used. They are written to a line of the register configuration files and affect the succeeding register file entries. The following modifiers exist:

- `->RegChannelOne`: Load first local register bank.
- `->RegChannelTwo`: Load second local register bank.
- `->RegChannelBoth`: Load both local register banks with the same data.
- `->RegGlobal`: Load global register bank.

The example below loads the FIR filter coeffcients of channel one, the peak finder condition of both channels and the global register `ReadoutEnable`.

```
->RegChannelOne
FIRCoeff1 -1
FIRCoeff2 0
FIRCoeff3 4
FIRCoeff4 0
FIRCoeff5 -1

->RegChannelBoth
PeakFinderCond 0

->RegGlobal
ReadoutEnable 1
```

6.3 Module Lists

For specifying the list of modules, which belong to a design and have to be processed for simulation and synthesis there exist some files, specifying lists of modules. They are contained in the subdirectory `modulelist` of the `verilog` and `netlist` directory. The files have the name of the top module to be simulated extended by the suffix `.ml`, e.g. `testPaBigTop.ml` for the simulation of the PPrAsic top module.

The module lists have to include the test bench and all test modules required by that. The same test bench and modules should be used for simulating at functional and netlist level, so the list should point to the same files for these module.

6.4 Simulation Makefile

The directory `verilog` contains a makefile used to generate simulation input. The following targets are available:

- `sim`: Make all what is necessary for simulation. This include making registers and stimulus files and running the Ptolemy simulation.
- `reg`: Make registers.
- `stim`: Make all stimulus files. This includes running the Ptolemy simulation.
- `serstim`: Make stimulus files for serial interface of PPrAsic.
- `ptolemy`: Run Ptolemy simulation.
- `html`: Generate HTML version of Verilog code.

6.5 Functional Simulation

The design entry is done with Verilog. The *Verilog-XL* simulator from *Cadence* is used to simulate the Verilog code. It is started from the UNIX shell with the command `verilog`. Prerequisite is that the script file `/usr/local/cad/scripts/cds4.42rc` was executed (preferably in the `.bashrc`). Documentation for the simulator can be found in *Openbook*, the Cadence online documentation. This is started with the command `openbook`.

One way to display simulation results is the use of a waveform display. The one included with Cadence is *SignalScan*. It can be started with the command `signalscan`. In order to get the data

from the simulation, the Verilog code has to contain statements to write a database of signals to disk. This is done by the following Verilog fragment which should be included into the top test module used for generation of the stimulus to the modules under test.

```
// Write data for waveform display
initial begin
  $dumpvars;
end
```

This will create a file `verilog.dump` which can be loaded by the waveform display.

For the PPrAsic two scripts have been created, to make the simulation process easier. For simulation use `veri <modulename>`. This reads from the directory `module_list` a file `modulename.ml`, which lists all files necessary to simulate the module, and runs *Verilog-XL*. To display the waveforms use `wadi`, which starts the waveform display and opens the database `verilog.dump`.

An alternative to Verilog-XL is *NC-Verilog*, which performs simulation by compiling Verilog to native code for the processor running the simulator. This can give much faster simulations than the interpretation approach of Verilog-XL. To use NC-Verilog use the script `ncveri <modulename>` instead of `veri`. To view the waveforms use `wave` as before.

See section 6.10 for an example of a simulation run. This includes all actions the user has to take.

6.6 Timing Simulation

6.7 Ptolemy Simulation of BCID

Before the details of the ptolemy simulation are explained it is neccessary to describe the structure of it. There are two ways to use the ptolemy tool. The first is to use the graphical interface which is helpful on the way to the final simulation. The second posibility is to create a script which runs directly from the shell. The first way was used to build the simulation environment which then was exported to several scripts with slight modifications for the different channels. These scripts are used to run the simulation independent from the graphical interface which is essential for the builtup of an automatic simulation system.

To be able to simulate the BCID with different settings without having to edit many files, which would make the system more susceptible for errors, a hierarchical script structure is used. Before a testrun is started the serial stimulus file has to be edited (see section 6.2). This defines the file where the register settings are stored (by the `LoadReg` command). In this file the registers for the FIR-filter, the BCID for saturated pulses and some other important registers are defined. This can be done separately for each channel For detailed information of the meaning of these registers it is referred to the *PPrAsic User and Reference Manual*.

To run the Ptolemy simulation use the command `make stim`. This generates the files required for the Ptolemy simulation.

6.7.1 Ptolemy model of BCID

To simulate the BCID in its full complexity a verilog-independent model of the BCID block on the PPrAsic is generated. The idea is to create a model that includes all the functional blocks of the BCID on the ASIC. This is done independent from the verilog code to be sure that problems based on the verilog desciption can be identified. The realisation is done with *Ptolemy*, because this program gives a good environment and supports a module based implementation. For each functional module, that the BCID block of the ASIC has a *star* in the ptolemy simulation is created.

These *stars* have the same functionality in the ptolemy simulation as their verilog counterparts. Figure 6.1 shows an overview of the top level of the PPrAsic and its environment in ptolemy. All what is described in following can be done separately for each channel. The files used have a channel number added to their names.

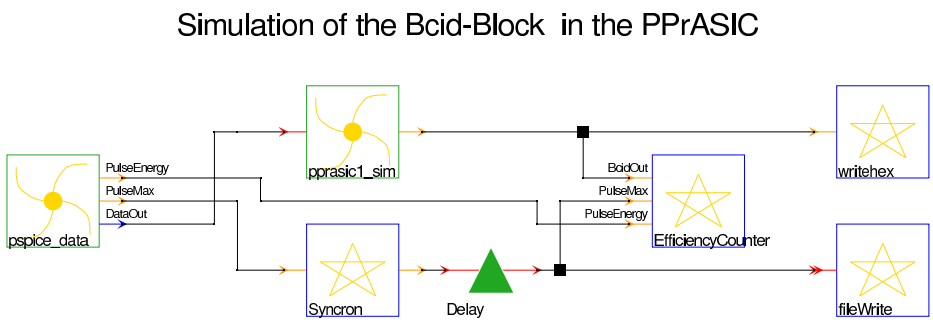

Figure 6.1: The overview of the ptolemy top-level module to simulate the PPrAsic

There are two different groups of modules in the figure. The blocks with the star are standing for modules with a certain fuctionality that is determined by a C++ based code. The other two blocks have an inner structure with modules of the same category that were explained before. In the following all these blocks will be described.

6.7.2 The PSpice data block

This block is used to make pulse data available for the following logic. Figure 6.2 shows its inner structure.

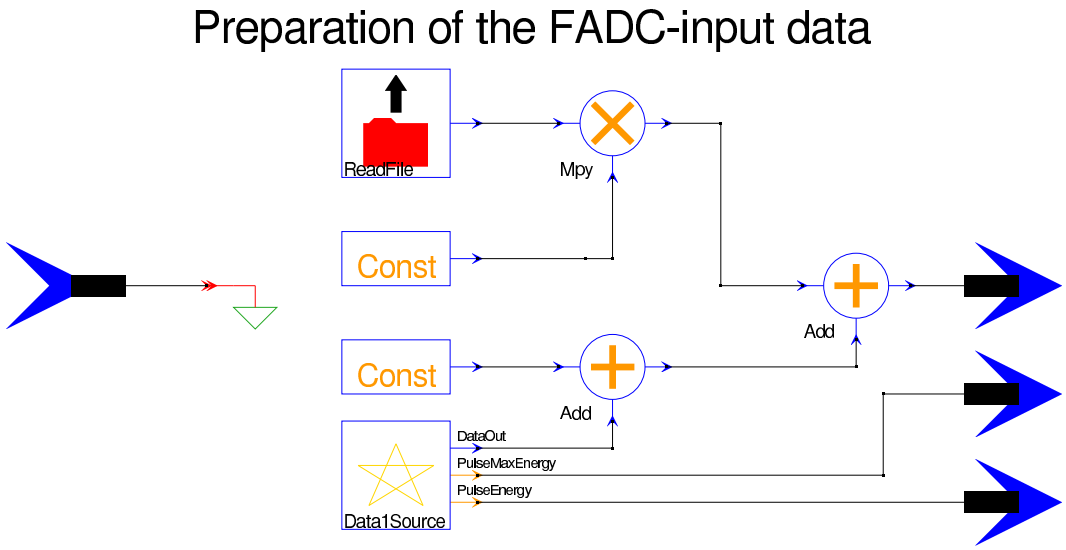

Figure 6.2: The overview about the block that reads in the PSpice input data and prepares them for the ptolemy logic

The ReadFile block is used to read in datas for the noise from the preamplification and the pileup from a file `stimulus/ptolemy/preamp_noise+pileup.dat`. The data in that file are created as described in [XXX]. They are multiplied by a constant that can be set by editing a file. All the files used to set constants for the ptolemy simulation that are counterparts to the programmable registers in the verilog code are listed and explained later. The Data1Source *star* is used to read in a file with analog pulse datas that were generated with *PSpice* and two additional files that are depending on it. These three files are `testdata.dat`, `PulseEnergy1.dat` and `PulseMax1.dat`, all located in the subdirectory `stimulus/ptolemy/`. The `testdata.dat` is generated from the outputfiles of the PSpice simulation in the following order. The output of the PSpice simulation is an enormous file which has all the information of the process and the resulting analog pulse datas for the pulses in the energy range that was set. To reduce this amount of data to a file which has only the analog pulse energies one after the other a *Perl* script `PSpiceTOptolemy` is used. It also creates a file which list the energies belonging to the pulses `PulseEnergy.dat`. These energies are taken from the PSpice number corrected by a factor (multiplied with 0.84875). The third file that is needed, `PulseMax1.dat`, has to be written directly. It has to imply three informations: the length of each pulse (*ns*), the time for each pulsmaximum to occure after the begining of the pulse (*ns*) and the number of pulses in the `testdata.dat` file.

6.7.3 The PPrAsic simulation block

This block represents the complete simulation of the BCID block in the PPrAsic. It gets the input data from the PSpice simulation and digitizes them. The results are used internally to produce the BCID output. To produce a file that can be read in by the verilog simulation the *writehex star* is used. The *Bitinversion* module is used to invert the MSB. This is necessary because the verilog code is adapted to a special FADC that inverts the MSB at its output. The simulation is also separated into logic modules that are the direct counterparts to the verilog modules. The external BCID bit is represented by a constant value that can be set. Figure 6.3 shows an overview about the structure.

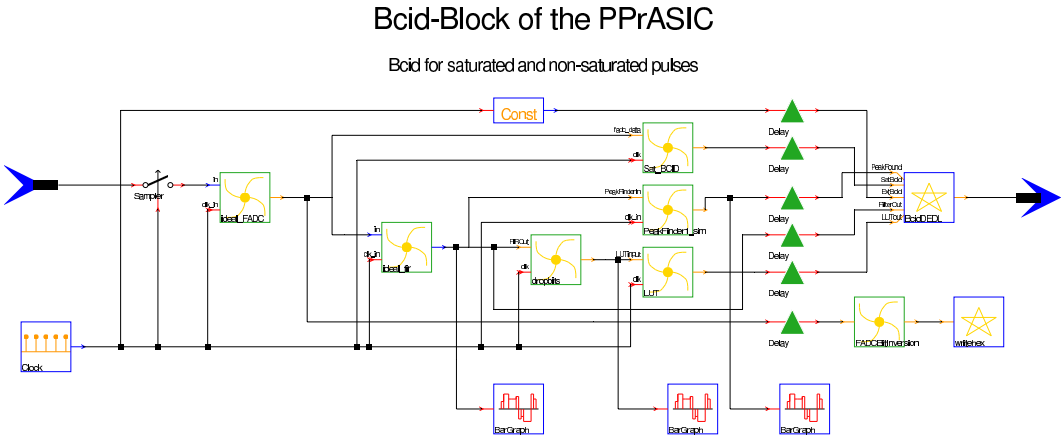

Figure 6.3: The overview about the part into the ptolemy simulation that represents the BCID block in the verilog code

6.7.4 The Syncron module

The Syncron module gets the information about the real peak position. This information is read in with an accuracy of 1 ns, but the BCID output data are just coming in steps of 25 ns. To be

able to compare these two data the peak position with the 1 ns resolution has to be fitted correctly on the timing scale with 25 ns steps.

6.7.5 The Efficiency Counter

The Efficiency Counter gets as an input results of the ptolemy simulation and compares them with the timestep of the real pulsemaximum. With the third information about the real energy of the pulse this *star* is able to produce a graph that shows the efficiency of the chosen algorithm depending on the pulse energy. The spare modules *fileWrite* and *writehex* are used to write data into files. The *writehex* module writes out the BCID results of the ptolemy simulation that are read in by a verilog module `verilog/tbPtolemyVerilogComp.v`.

6.8 Comparison of Ptolemy and Verilog simulation

To be able to compare the results of the Ptolemy and Verilog simulation a verilog module called `tbPtolemyVerilogComp` exists. It is used in the testbench. This module gets as input data the results of the Verilog simulation of the ASIC and the results of the Ptolemy simulation, which are read in from files called `SimOut1.out` and `SimOut2.out` out of the Ptolemy directory (`pprasic/stimulus/ptolemy`). It compares this two data sets and prints out the number of matches and errors.

To create these data and the input data for the ASIC a Ptolemy simulation has to be done. For this you have to make sure that all files Ptolemy needs do exist. You might have to run a dummy `make serstim` loading registers and LUTs. Then you have to modify the configuration file for the ASIC registers to get the configuration you want to test and then type `make stim` as a shell command. This will run the Ptolemy simulation and create the result files for the comparison.

Before the Verilog simulation is started it has to be defined when the comparision will start. Therefore you have to set the *PtolemyCompare* input of the module `tbPtolemyVerilogComp` to high. It should be done after all the configuration and the LUT data are read into the ASIC to avoid errors. It is done by adding the comand `Verilog PtolemyCompare=1;` to the ser.stim file.

To be able to run various simulations one after the other a perl-script was written. This script `checklist_simulation` runs the first 24 tests that are defined in the `CHECKLIST` and writes the results into a file called `ChecklistResults.dat`. This script can be used to check the Verilog code as well as the netlist. The running of the script is done by typing `checklist_simulation` as a shell command.

6.9 Simulation Logging

For logging of simulations runs a script `simlog` is provided. It stores all relevant data required to redo the simulation and simulation results to a file located in the global directory `simlogs` (see 2.2 and makes an short entry in the master log file `inex.modulename.log` located in the same directory. The `modulename` part is replaced by the top module used in the simulation to be logged. Additional informations like a summary and description of the simulation run are queried from the user.

The `veri` and `ncveri` scripts automatically call the `simlog` script after finishing the simulation.

6.10 Example simulation run

This section gives an example of a simulation run. The user-supplied commands are indicated by the shell-prompt `shell>`.

```
shell> make reg
/cad5/caduser/huebner/bin/makereg PPrAsicChannelReg.conf PaChannelReg \
             PaChannelReg.v IncPaChannelReg.v ../stimulus/PaChannelReg.pl
(...)
shell> make serstim
cd ../stimulus; make_serstim ser.huebner.stim ser.v
Wait 0xa
LoadReg init.reg
Register values for channel 1:
FIRCoeff1 => 0xf
FIRCoeff2 => 0x0
FIRCoeff3 => 0x4
(...)
shell> ncveri testPaBigTop
ncxlmode: v1.22.(s39): (c) Copyright 1995 - 1997 Cadence Design Systems, Inc.
(...)
18a3:  Data  0a3                          Time: 2624400
18a4:  Data  0a4                          Time: 2624725
0000:                      RB Header 000 Empty   Time: 2625050
1057: Header 057                          Time: 2625375
1800:  Data  000                          Time: 2625700
18d2:  Data  0d2                          Time: 2626025
18d3:  Data  0d3                          Time: 2626350
18d4:  Data  0d4                          Time: 2626675
Number of not identified peaks in the first channel:  2054
Number of identified peaks in the first channel    :     1
Number of not identified peaks in the second channel:  2054
Number of identified peaks in the second channel    :     1
Memory Usage - 3.5M program + 70.1M data = 73.5M total
CPU Usage - 4.8s system + 491.4s user = 496.2s total (98.7% cpu)
Simulation complete via $finish(2) at time 2626975 NS + 1
./tbPtolemyVerilogComp.v:108        $finish(2);
ncsim> exit
shell> wadi
shell>
```

Chapter 7

Timing Analysis

Timing verification is performed using the *Pearl* static timing analyzer. Timing analysis could be done on the netlists generated both before and after layout.

7.1 Pre-Layout

Pre-layout analysis could be done on the top level of the PPrASIC by first generating a delay file in the Standard Delay Format using the *Pearl* command file `pearl.cmd`, which is a pearl script. A typical *Pearl* command file could for instance contain:

```
SetVoltage 3.0:3.3:3.6
SetTemperature 0:25:70
SetProcess 0.8:1.0:1.2
ReadGCFTimingLibraries ./cup3.3V.gcf
ReadVerilog ./PaBigTop.v
TopLevelCell PaBigTop
# InputSlew * 0 5 0 5
WriteSDFDelays PaBigTop.sdf
quit
```

The GCF file contains operating conditions and reads timing libraries for the design and memory blocks. In the pre-layout case the parasitic data, interconnect and cell delays are the estimated ones. The generated delay file, i.e. `PaBigTop.sdf`, is then used, together with the original netlist, to do timing analysis. This type of timing verification, although not accurate, produces first comparison between *Pearl* and synopsys timing report, and could reveal eventual errors or inconsistencies in the design at an early stage. An example of a possible script to start an interactive timing analysis with *Pearl* could look as follows:

```
# open Log file
LogFile PaBigTop.log
# read in technology/ram timing libraries (CTLF format)
ReadGCFTimingLibraries cup3.3V.gcf
# read in design (PaBigTop)
ReadVerilog PaBigTop.v
TopLevelCell PaBigTop
# read in delays
ReadSDF -all PaBigTop.sdf
```

```
# delay annotation summary
ShowDelayAnnotationSummary -show_all
# define clock signals (timing constraint)
#     -----period-----     node      rise     fall
Clock -cycle_time 25.0       Clk     12.5     25.0
Clock -cycle_time 12.5    SerClk      6.25    12.5
Clock -cycle_time 100.0  JtagTck     45.0     55.0
# find longest paths (first SMPs) triggered by posedge Clk
SetPathFilter -same_node
FindPathsFrom Clk ^
ShowPossibility
# verify design timing
TimingVerify
ShowPossibility
```

At this stage the user is under *Pearl* and could continue timing analysis using different *Pearl* commands.

7.2 Post-Layout

Post-Layout timing verification is the preferred method to do final design checks. Here the parasitics data and the interconnect delays are exact ones generated by *Cadence*. The delay file generated in this case is also in Standared Delay Format. In addition the generated clock tree, containing buffers in the clock networks, is also included in this SDF file. A final netlist, based on the delay information, is also produced in *Cadence*. This verilog netlist file is different from the one generated by *Design Compiler* in that it also contains information concerning the clock tree. Here one can also use the same *Pearl* command file from the previous section. The only difference is that here the netlist and sdf files are the post-layout ones.

Chapter 8

Documentation

The design of the PPrAsic is based on a specification document [9]. This document describes the function of the PPrAsic in a general way. More detailed information can be found in two other documents describing the PPrAsic and its design. They are located in the `doc` directory of the PPrAsic CVS module. This directory has two subdirectories. `manual` contains the *User and Reference Manual* and `designguide` contains the designguide you are reading now. Both are LaTeX documents. The top-level files are `pprasicman.tex` for the manual and `pprasicguide.tex` for the designguide. All documents are available on the web [11].

In addition to these documents source code documentation is essential for a successful design. To avoid inconsistencies the Verilog source code includes this documentation. For each module the purpose and interface should be described. The function of the I/O signals should be documented, including relations to other signals, timing, meaning of different values, etc. A convenient way to view the source code is provided by conversion to HTML code, which is available on the web [4].

Chapter 9

Test strategies

It is important to simulate all components and the whole design carefully and thoroughly in order to ensure that the chip will meet all requirements. This includes functional simulation, simulation with gate and path delays and analysis of all aspects not covered by simulation, i.e correct clock distribution, sufficient power supply, consideration of external constraints (delays, fan-out, etc.), design rule check (DRC), layout vs. schematic check (LVS) and further checks (static timing analysis, test coverage analysis, etc.).

Another important topic is the test of the chip after production. This can be done with the HP82000 chip tester using test vectors from the simulations. This includes the use of vectors for expected output data.

9.1 Simulation

Simulation is done in three stages. First stage is functional simulation, which is used to debug the Verilog code and to ensure that the logical function of the description is correct and meets the reuqirements. The second stage is simulation after synthesis, which verifies that the synthesized circuit is equivalent to the functional description. The third stage is simulation with full delays which is used to verify that no timing problems arise and the circuit meets the timing requirements (minimal system clock frequency, etc.).

The following aspects have to be considered for simulation:

1. The test vectors should cover as many states of the circuit as possible. Special care should be taken for simulation of typical conditions under which the chip will be operated, boundary conditions like minimum and maximum values of counters, adders and multipliers, power-up etc. and some sort of arbitrary or random inputs to find unexpected errors.
2. Compare the results of the simulation to results obtained independently. Avoid making the same error twice, once in the code you simulate and another time in the interpretation of your result. To prevent this, it is useful if the simulation is repeated by another person than the designer of the block under simulation.
3. All three stages of simulation should be done with the same test benches and vectors. The results must match.
4. Simulation results should be documented.
5. Most important are simulations of the complete design, but even trivial blocks should be simulated, preferably independently of the rest of the design. Verilog code not simulated is wrong.

9.2 Chip test

The test of the produced chip is done with the HP82000 chip tester. Test vectors and vectors for expected output data are generated automatically from the simulation. The same rules for test coverage apply for the chip test as for the simulation of the design. This approach allows to verify very quickly if the behaviour of the real chip matches the behaviour of the simulation, which has to be ensured to meet the specifications.

9.3 JTAG

The JTAG interface provides a way to access the internals of the circuit through an independent path using only a few signals. Internally scan paths can be connected to the interface.

9.3.1 Boundary scan

Boundary scan connects a scan path including all pads of the chip to the JTAG interface. This allows to set all output pads to defined values and to read all input pads. This is used to check the interconnectivity between the chip and its environment.

9.3.2 Internal scan paths

Some or all flip-flops in the design can be replaced by special scan path flip-flops, which can be connected to one or more scan path. These scan paths can be used to set registers in the design to certain values or to read back the stored values using the JTAG interface.

Chapter 10

Design Reviews

For the success of a design it is very useful if it can benefit from the expertise of other people not directly involved in the design. For this reason the design is reviewed at different stages by external experts in order to find problems the designer might be "blind" for.

For the PPrAsic three reviews are planned. Two of them are defined by the ATLAS project and an internal review by ASIC laboratory experts is added. At the beginning of the PPrAsic design the specification is subject to a *Preliminary Design Review* (PDR) involving people of the ATLAS Level-1 trigger group. Shortly before finishing the design an internal review by members of the Heidelberg ASIC lab not belonging to the ATLAS group is performed. After that, before the chip is finally submitted for production, a *Final Design Review* (FDR) takes place, involving a smilar group of people as the PDR.

This procedure should allow to identify all problems related to the PPrAsic design before a substantial amount of money or time is spent on faulty chips.

Bibliography

[1] Pre-Processor Asic, User and Reference Manual,
http://wwwasic.kip.uni-heidelberg.de/atlas/projects/pprasic.html

[2] Openbook, Cadence Online Documentation

[3] Sold, Synopsys Online Documentation

[4] PPrAsic Verilog source code,
http://wwwasic.kip.uni-heidelberg.de/atlas/DATA/verilog/pprasic/hierarchy.html

[5] Web-Interface to PPrAsic CVS repository,
http://wwwasic.kip.uni-heidelberg.de/cgi-bin/cvsweb.cgi/pprasic/

[6] Hardware Diagnostic, Monitoring and Control Software (HDMC),
http://wwwasic.kip.uni-heidelberg.de/atlas/projects/hdmc.html

[7] HDMC Online Help,
http://wwwasic.kip.uni-heidelberg.de/atlas/DATA/source/hdmchelp

[8] The Ptolemy simulation system,
http://ptolemy.eecs.berkeley.edu

[9] PPrAsic Specification,
http://wwwasic.kip.uni-heidelberg.de/atlas/L1/DISCUSS/PPrA_SpecJul99.ps

[10] ATLAS First-Level Trigger Technical Design Report, ATLAS TDR-12, CERN/LHCC/98-14, CERN, 24 June 1998,
http://atlasinfo.cern.ch/Atlas/GROUPS/DAQTRIG/TDR/tdr.html

[11] Documentation page of the Heidelberg ATLAS group,
http://wwwasic.kip.uni-heidelberg.de/atlas/docs