Code Generation from a Single Source Structural Specification

Claus Schneider, Wilfried Gänsheimer, Erich Schwenk
Micronas GmbH
Frankenthalerstr. 2
81539 Munich, Germany
<first>.<last>@micronas.com

ABSTRACT
A method and a tool to automatically derive top-level hardware description, low-level driver software and documentation out of a single source structural specification is presented. In contrast to system-level design tools this approach focuses on structural properties like interconnect, hierarchy and control/status registers instead of refining a functional specification. A table-based specification format was chosen to enable efficient data entry and review. Due to automatic code generation consistency between specification and design is guaranteed and error-prone manual transformations can be avoided. This paper focuses on specification and hardware generation.

Keywords
Single Source, Structural Specification, Code Generation, Table, Spreadsheet

1 INTRODUCTION

System-level design is a highly interactive process. A huge amount of experience is needed and many trade-offs have to be made during design space exploration. Therefore, it is extremely difficult to automate this process, and most of the refinement steps from specification to design have to be done manually. Depending on the purpose of a model different languages may be used. In reality an executable specification is only available for parts of the design. These models were designed to perform algorithm evaluation and may serve as reference models for verification, but in most cases they are not suitable for direct refinement to a design realization. In addition, initialization is done differently in the abstract model than in the concrete design, and IO and test structures are not covered at all.

Due to these constraints we decided to focus on structural properties like interconnect, hierarchy and control/status registers instead of refining a complete functional specification.

In this approach, functional blocks are treated as black boxes for top-level integration. The control and status registers are separated from the functional block to enable different address mappings and integration into different chip architectures (e.g. parallel control bus vs. serial control channels). Hierarchy information is separated from interconnect specification to enable easy hierarchy changes and flat (source to sink) interconnect analysis across hierarchies. Another structural issue is the multiplexing of a huge number of internal signals to a limited number of external IO pads.

A tabular specification format was chosen for top-level interconnectivity, hierarchy, IO multiplexing and control/status registers. The major benefit of this approach is to maintain a single source of specification information. All code and documentation that can be directly derived from the specification is generated automatically to avoid any error-prone manual interaction and to guarantee for consistency between specification, documentation and design.

2 RELATED WORK

The proposed table-based structural specification is closely related to traditional text-based or schematic design entry but is intended as a method to bridge the gap between concept and design. In the first step the concept engineering captures the register specification and basic architecture definition (major control and data flow). Afterwards the specification is extended by implementation details to be ready for automatic code generation.

Concept engineers prefer spreadsheets whereas designers often stick to their favorite text or schematic editor to capture the design. For this reason, the major issues related to these methods will be discussed briefly. While text-based design entry requires focus on syntax and formatting in schematic entry much time is spent on doing layout of blocks and wires. Both methods require a significant amount of effort, which is not necessary to reach the goal – a compact specification of logical interconnect. On one hand block diagrams are very useful to understand and to discuss structural design issues but on the other hand they are very time consuming to create and to maintain. A solution to this problem is the creation of HDL code out of a compact logical interconnectivity specification and the visualization with generated block diagrams.

Meanwhile, sophisticated debug tools [1] generate acceptable hierarchical block diagrams and allow analysis of fan-in and fan-out cones of signals across hierarchies. Unfortunately, the analysis capabilities are restricted to debugging and tracing of single signals. They are neither intended to a systematic review of all top-level interconnections nor for design entry and code generation.

The table-based capture of structural specification information fills this gap between concept and design. It provides a means for a compact interconnect specification without having to deal with HDL syntax nor schematic layout issues. The HDL code is generated automatically according to predefined coding guidelines and naming styles. Schematics can be generated dynamically on demand by commercial tools.

The hierarchy that is required by text and schematic entry to handle the complexity is kept separate from the interconnect definition. Filter capabilities in spreadsheets instead of hierarchy can be used to deal with complexity. By keeping hierarchy separate from interconnect it can be changed easily and signals can be traced at a flat top-level from source to all sinks across hierarchies. While text and schematics are block-oriented the table provides also an interconnect-oriented view.

A commercial interface-based design tool [2] provides a table-based interconnect entry but there is no separation of interconnect and hierarchy nor there is dedicated support for register and IO multiplex specification. Especially the support for register, IO, hierarchy and interconnect specification in a single source for automatic generation of derived code and documentation is seen as the key benefit of the proposed tabular specification.

3 MAPPING OF SPECIFICATION OBJECTS TO DESIGN INSTANCES

The single source structural specification consists of four tables (Interconnect, Hierarchy, IO, Register). Out of these tables, top-level hardware structure, HW/SW interface (register structure + low-level driver software) and documentation are generated.

3.1 Software and Documentation

The lowest layer of the driver software is generated out of the register sheet only. The purpose of this layer is to hide the mapping of logical control/status parameters to physical bit slices of one or several registers. The higher software layers can access these control parameters without having to care about address space mapping and masking operations. For documentation purposes various tables are generated that list registers ordered by address, alphabetically or in functional order. In addition an IO mapping table is generated to summarize the IO multiplex options.

This paper focuses on the generation of the top-level interconnectivity. The basic architecture shown in Figure 1 will be taken to explain the method.

3.2 Basic Target Architecture

Typically, our chips are structured in the following way:

- CHIP: Pad frame containing the pad instances
- TOP: IO control Cells (IOC) that perform multiplexing, grouped into N/S/W/E for layout purposes
- CORE: Subsystems consisting of functional blocks, registers and interface blocks. For mixed VHDL/ Verilog designs wrapper blocks are necessary.

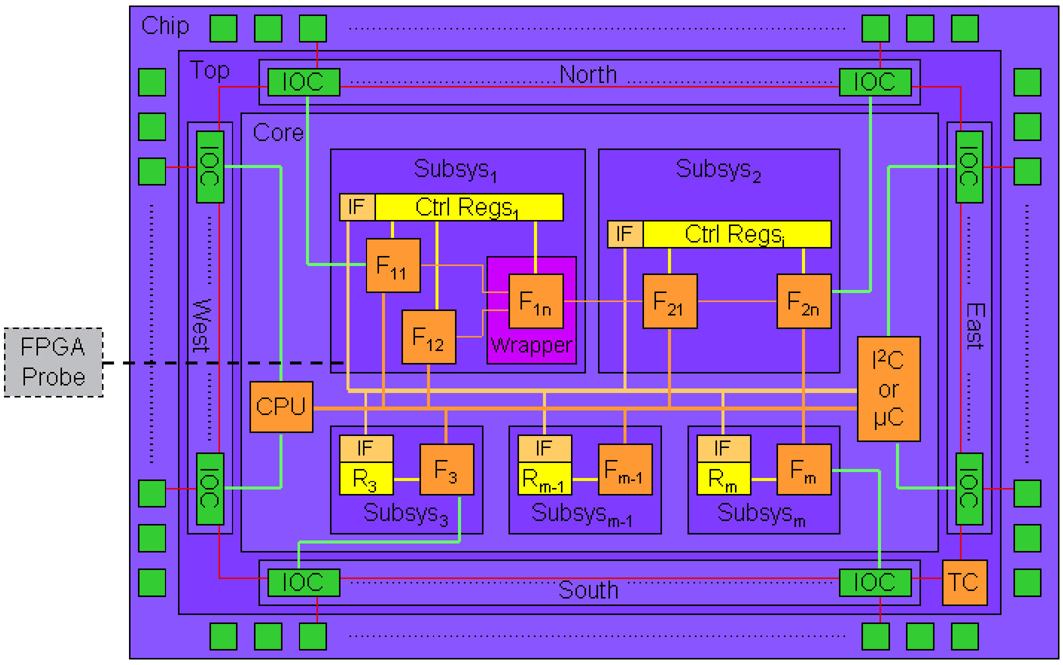

Figure 1: Basic Target Architecture

Control-oriented designs may consist of an internal CPU and a micro controller, both with their own bus. Designs for signal processing may only consist of a dedicated data path and a serial programming interface (e.g. I^2C). A test controller (TC) and a chain connecting all IOCs (e.g. boundary scan or NAND tree) denote the presence of test logic.

Not all connections have to be specified explicitly in the interconnect table for the generation of the top-level interconnectivity. Connections are automatically generated between core signals and IOC blocks by referencing signals from the interconnect table inside the IO table.

An example is the connection between F_m and IOC (green). Additionally, so-called implicit signals are generated by assignment of a register to a functional block in the register table, e.g. the connection between R_m and F_m (yellow). All other connections have to be specified explicitly (orange). User defined macros, e.g. for signal bundles, are supported to increase the abstraction level of the specification. Further reduction of typing can be achieved for regular interconnect structures, like boundary scan chains, by the usage of regular expression matching for instance names (red). These features will be described in more detail in the following sections.

All interconnections are described directly between functional blocks, registers, IO cells and pads. These blocks are treated as black boxes for top-level integration. The methodology can be applied in a hierarchical manner because there is no assumption on the granularity of the leaf blocks. A module designer may treat adders, FIFOs or FSMs as leaf blocks. For chip-level integration these internally developed modules or intellectual property blocks from third parties may be treated as leaf blocks (black box). Intermediate hierarchies, e.g. CHIP, TOP, CORE and subsystems, or simulator dependent wrapper frames between VHDL and Verilog are not considered for the interconnectivity specification. The hierarchy is specified in a separate table and used only for HDL code generation. While hierarchy is required for text or schematic capture it is replaced by filter capabilities of spreadsheets. Therefore, several attributes, assigned to the signals, can be used to focus on a selected aspect of the design. Besides a bundle name to group signals a class attribute is provided to select between data, control, test, clock or reset signals. This capability is especially useful for systematic reviews: All signals that belong to a certain category, e.g. test, can be traced from source to all destinations across all hierarchies. This can be done in a systematic manner rather than randomly one signal after another in a schematic tool.

3.3 Interconnect

There are four different types of interconnect specification objects that directly correspond to VHDL or Verilog design objects. A signal, in Verilog called net, is a connection between leaf blocks that has at least one driver (source) and at least one load (destination). A port describes an interface of the top-level and can be of mode *in*, *out* or *inout*. Constants are used to parameterize instances. Generics provide a means to pass constant parameters to leaf modules. The basic source of interconnect specification is the interconnect table, a list of interconnect objects. An example is shown in Figure 2.

The first row holds the tags identify the columns. This allows for swapping or inserting columns without the need to modify the table parser. There are two additional columns that are not shown in Figure 2: There is a *description text* column and an *ignore* column that can be used to mark a row as a comment line. A mode attribute distinguishes between different objects: Signal, port,

::gen	::bundle	::class	::clock	::type	::high	::low	::mode	::name	::out	::in
	RGB	Data	CLKF	logic	7	0		SIG_R	MODA/R	MODC/R
	RGB	Data	CLKF	logic	7	0		SIG_G	MODA/G_LO(3:0)=(3:0), MODB/G_HI(3:0)=(7:4)	MODC/G(5:1)=(7:3), MODD/G(7:5)=(2:0), MODE/G=(0)
	RGB	Data	CLKF	resolved	7	0		SIG_B	MODA/B(7:0), MODB/B(7:0)	MODC/B(7:0)
$i (1..147), /IOC_(\w+)_$i/	NAND_TREE	Test	TCK	logic				NAND_OUT_$i	IOC_$1_$i/NAND_OUT	IOC_$1_{$i+1}/NAND_IN
/IOC_(\w*O)_(\d+)/	PAD_CTRL	IO		logic				PAD_DO_$2	IOC_$1_$2/DO	PAD_$2/DC
MH		StdIF	$1	YCrCb_$2_$3_$4					$5	$6
MD	$5_$6	StdIF C	$1	logic				$5_AP	$5/AP	$6/AP
MD	$5_$6	StdIF C	$1	logic				$5_HSYNC	$5/HSYNC	$6/HSYNC
MD	$5_$6	StdIF C	$1	logic				$5_VSYNC	$5/VSYNC	$6/VSYNC
MD	$5_$6	StdIF D	$1	logic	$4	0		$5_CB	$5/CB($4:0)	$6/CB($4:0)
MD	$5_$6	StdIF D	$1	logic	$3	0		$5_CR	$5/CR($3:0)	$6/CR($3:0)
MD	$5_$6	StdIF D	$1	logic	$2	0		$5_Y	$5/Y($2:0)	$6/Y($2:0)
MX		StdIF	CLKD	YCrCb_8_7_7					BLE	CST
				integer			C		6	FIFO/DEPTH
/(IOC_\w+_\d+)/	TOP	IF		integer			G	WIDTH	8	$1/WIDTH
	TOP	IF	CLKF	logic	WIDTH-1	0	I	DATA_IN		MODC/Y

Figure 2: Interconnectivity Table

generic and constant. The name of each interconnect object has to be unique for the flat interconnect specification. Objects are further described by a type and range (for vector types), bundle name and class attribute for sorting and filtering purposes and an associated clock for signals and ports. At the beginning of the table different examples for signal definitions are shown. In the single bus example the width of the ports can be omitted when identical to the signal width. The split bus example shows a signal where one bit slice is driven from module A while the second slice is driven from module B. At the destinations the signal branches to modules C, D and E. For this connection the bits 7 down to 3 of the signal are mapped to bits 5 through 1 of the port G of module C. Tri-state bus connections can also be specified. The usage of regular expression matching and a counter loop for a very compact specification of a regular interconnectivity structure is shown for a NAND tree.

The algorithm to generate the regular inter-connectivity structure is shown in Figure 3 in PERL syntax.

```
foreach my $i (1..147) {
  foreach $inst (%inst_hash) {
    if ($inst =~ /IOC_(\w_\w+)_$i/) {
      &add_conn(NAND_OUT_$i,
                IOC_$1_$i/NAND_OUT,
                IOC_$1_{$i+1}/NAND_IN, .... )
    }
  }
}
```

Figure 3: Regular Expression Matching

For each instance matching the regular expression within the specified loop a connection is added. The variables $1 (matched part of instance name), $i (loop counter) and any expression like {$i+1} will be evaluated and replaced by their actual value. For simplification the evaluation is not shown in Figure 3. After the NAND Tree an example for multiple point-to-point connections of the data out ports between all IO control cells and pad cells is shown.

For standardized interfaces like micro controller busses or data path connections macros can be defined to reduce typing effort and support abstraction. In Figure 2 a data path example is shown consisting of control signals (AP, HSYNC, VSYNC) and data signals (CR, CB, Y). The clock parameter ($1) of the macro header (MH) is just copied to all signals of the bundle in the macro definition (MD). The data widths ($2, $3, $4), the source ($5) and destination ($6) blocks will be replaced in the macro execution (MX) by the actual values.

Finally, a constant definition to parameterize a FIFO depth, a generic definition to parameterize the width of IO cells and an input port whose width depends on the generic parameter are shown in Figure 2.

3.4 Hierarchy

Interconnectivity and hierarchy are separated to enable flat analysis of top-level interconnectivity. For each level of hierarchy all instances are listed with a reference to the parent hierarchy. The most important instance attributes like entity name, configuration name and language (VHDL or Verilog) are specified, Figure 4.

In the hierarchy table regular expressions for instance matching can be used for grouping of instances. This feature is shown for pads and IO control cells. In the PADS hierarchy all instances matching “PAD_” followed by a number between zero and 147 are grouped. These instances were generated from the IO table described in the next section. The IO table also defines the pad type, i.e. the entity name. To avoid duplicated information and to enable grouping by regular expression matching the entity table cell for pad instances is left empty. The generator tool gets the correct entity name for each pad instance from the IO table to expand the hierarchy table. A similar mechanism applies to the IO control cells. Here, the tool automatically generates an entity name according to the given naming rules.

Another feature of the hierarchy table is the specification of design variants. A special column (::variant) in the table is reserved for a list of variant names for each instance. Using this feature the PROBE module can be included in the FPGA variant for debugging purposes and can be dropped in the final Chip (Default) or Emulation version. We also used variants to replace various data path modules by bypass modules to reduce the gate count for the FPGA prototype. In this scenario modules that were not relevant for some tests were bypassed.

::gen	::variants	::parent	::inst	::lang	::entity	::config
	Default	TESTBENCH	MYCHIP	VHDL	MYCHIP	MYCHIP_RTL_CONF
	FPGA	TESTBENCH	PROBE	VHDL	PROBE	PROBE_RTL_CONF
	Default	MYCHIP	PADS	VHDL	PADS	PADS_RTL_CONF
	Default	MYCHIP	IO_SOUTH	VHDL	IO_SOUTH	IO_SOUTH_RTL_CONF
	Default	MYCHIP	TOP	VHDL	TOP	TOP_RTL_CONF
$i (1..147), /PAD_$i/	Default	PADS	PAD_$i	VHDL		${::entity}_structural_conf
$i (1..42), /IOC_(\w_\w+)_$i/	Default	IO_SOUTH	IOC_$1_$i	VHDL		${::entity}_RTL_CONF

Figure 4: Hierarchy Table

3.5 IO Structure

Most of the designs are pin-limited and require IO multiplexing in mission mode as well as in test mode. While multiplexing options can be described very efficiently in tabular format the manual HDL implementation of all interconnections is very inflexible, time consuming and error-prone. The automation of this task is based on predefined IO Control cells (IOC). Different types of interfaces (e.g. GPIO, DRAM) require different IOCs.

An example for an IOC is shown in Figure 5.

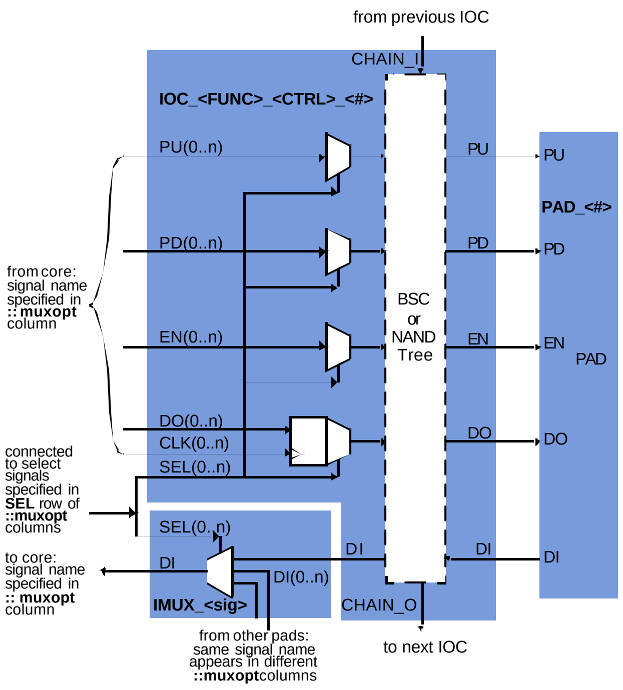
from previous IOC
CHAIN_I
IOC_<FUNC>_<CTRL>_<#>
PU(0..n)
PD(0..n)
EN(0..n)
DO(0..n)
CLK(0..n)
SEL(0..n)
from core: signal name specified in :: muxopt column
connected to select signals specified in SEL row of ::muxopt columns
SEL(0..n)
DI
DI(0..n)
IMUX_<sig>
to core: signal name specified in :: muxopt column
from other pads: same signal name appears in different ::muxoptcolumns
BSC or NAND Tree
PU
PD
EN
DO
DI
CHAIN_O
to next IOC
PAD_<#>
PU
PD
EN
PAD
DO
DI

Figure 5: IO Control Cell

The IOC contains multiplexing logic, output registers and it may contain also test logic like a NAND tree or a boundary scan chain. Regular interconnect structures like the NAND tree and the connections between IOC and pad can be explicitly described in the interconnect sheet using regular expression matching and loops. All connections of core signals to the IOC multiplexer inputs have to be described in the IO table. A small example of an IO table is shown in Figure 6. It defines an input pad TM, two supply pads (VDD, VSS), an IO pad (GPIO) and another IO pad with pull up (PCK). Each row of the table describes one pad, its attributes, the IO control cell instance and the IO multiplexing.

::pad	::type	::iocell	::port	::name	::muxopt	::muxopt
			SEL	PAD	IOSEL_0	IOSEL_T
1	WC3I80	IOC_G_I	DI	TM	TESTMODE	TESTMODE
2	WVV3IO			VDD		
3	WVV0IO			VSS		
4	WC3B60	IOC_R_IO	DI, DO, EN	GPIO	GPIN, GPOUT, GPEN	SCAN_IN
5	WC3BC0	IOC_R_IOU	DI, DO, EN, PU	PCK	, PCK, PCK_EN	TCK, , , '1'

Figure 6: IO Table

For the multiplexing a list of ports (::port) of the IOC is assigned to multiple lists of core signals (::muxopt). The direction of the pad is determined by the ports of the IOC, that are used in this multiplex option: Data In (DI), Data Out (DO), Output Enable (EN). In addition, Pull Up (PU) and Pull Down (PD) resistors in the pad can be controlled. A special row defines the control signals of the select inputs (SEL) of all IO multiplexers for each multiplex option (e.g. IOSEL_0, IOSEL_T). These select signals have to be explicitly defined in the interconnect sheet. The signals referenced in ::muxopt columns can be either explicitly defined signals from the interconnect table or implicitly defined signals that will be generated or expanded by the tool (e.g. signals from/to register blocks).

3.6 Register

Control and status registers are separated from the functionality of the processing units. This approach often referred to as interface-based design [3] or separation of synchronization and functionality [4] gives the maximum flexibility to adapt the units to different on-chip communication structures like control busses (e.g. µC) or control channels (e.g. ::type = I2C). An example of the register table is shown in Figure 7. Each row of the table defines the bits (::b) of a control parameter and it's attributes. The generator supports different access modes (::rw), auto address increment (::auto), update mechanisms (::sync) and reset values (::init). Several control parameters can be merged to a register by using the same sub address (::sub). Furthermore, registers can be grouped into register blocks (::interface). The grouping may be based on clock domains, update domains, or logical function. The HDL code generation works similar to the approach for the IO control cells. Different types of predefined registers with various generic parameters are instantiated and

::type	::sub	::interface	::block	::rw	::auto	::sync	::clock	::reset	::b	::b	::b	::b	::b	::b	::b	::b	::init	::view	::comment
I2C	30	I2C_VIDEO	RGBF	W	Y	VS_F	CLKF	RSTF			BRT.5	BRT.4	BRT.3	BRT.2	BRT.1	BRT.0	20	Y	Brightness Adjust
I2C	30	I2C_VIDEO	RGBF	W	Y	VS_F	CLKF	RSTF		DIGSEL							0		Select digital or analog input
I2C	30	I2C_VIDEO	RGBF	W	Y	VS_F	CLKF	RSTF	YUVSEL								1	Y	Select YUV or RGB input

Figure 7: Register Table

parameterized according to the register attributes defined in the table including clock and reset signals. The control parameters of the registers are connected to the functional blocks (::block) that are specified in the table. These are implicit connections and the corresponding ports at the functional blocks are automatically added. During this generation step parameters that are spread over multiple registers are combined to one signal. The hierarchical register blocks are also automatically generated. Finally, the register table includes attributes (::view, ::comment) for driver software and documentation generation.

4 IMPLEMENTATION

4.1 Data Model

The ***M**icronas **I**nterconnect Spec e**X**pander* converter tool (**MIX**) closely models the required high-level design objects like *instances*, *signals*, *IO cells* and *registers*. *Perl* provides powerful regular expression matching operators and text processing capabilities. Design objects and other data structures are mapped to hashes and arrays. The relation between different design objects is shown in an UML-like format in Figure 8.

A chip consisting of instances and signals can be seen as an abstract class. Only a concrete variant can be generated. Instances and signals are linked by ports. Hierarchical blocks can be nested while leaf blocks are not further decomposed. A register block is hierarchical, consisting of registers. IOCs, pads and functional blocks are leafs in the hierarchy. There are two types of signals: Implicit signals are automatically generated out of the control parameter specification of the register definition. These are the signals between functional blocks and the bit-fields of the control/status registers. Explicit signals are directly specified in the interconnect sheet as plain signals or they are expanded from macros or regular expressions.

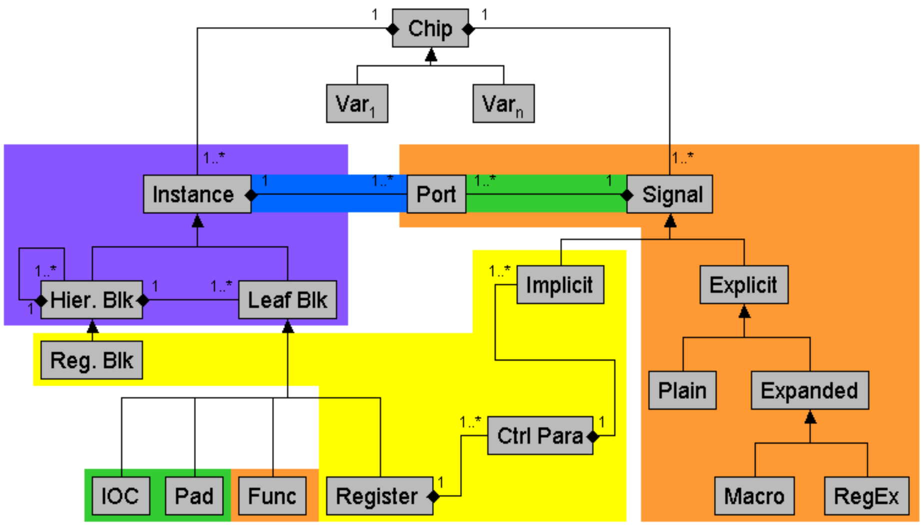

Figure 8: Data Model

4.2 Step 1: Read Input Description, Generate Intermediate Data

The converter tool starts up with some basic initialization. A set of company and project specific design guidelines and templates are read in and added to the tool environment. Design guidelines define standard naming conventions (e.g. pre- and postfixes) and other rules applied in later phases or used to check the generated data.

The input tables are checked to make sure all required columns are available. Default values are applied to empty cells if needed. Macro definitions and *Perl* match operators are preprocessed and stored in separate intermediate hash arrays to be applied later on.

The signal descriptions from the EXCEL tables are stored in the hash array `%conndb` modeling the non-hierarchical connection matrix CONN. The signal names are used as hash keys. Each array element itself is another hash array addressing the input data columns. Thus a signal and its properties can be accessed easily by its name. A unique, flat name space for signals is guaranteed. The signal `SIG_R` from the example in Figure 2 is stored as shown in Figure 9.

```
$conndb{'SIG_R'} = {
'::bundle' => 'RGB', '::class' => 'Data',
'::clock' => 'CLKF', '::type' => 'logic'
'::in' => [{'inst'=>'MODC', 'port'=>'R', … ],
'::out' => [{'inst'=>'MODA', 'port' => 'R', … ],
'::descr' => 'single bus', …};
```

Figure 9: Signal Data Structure

Macro expansion helps to specify regular or repetitive connections like standard on-chip buses very efficiently. The CONN data structure gets extended when additional information for a signal is defined in later stages.

For the instance specification a similar approach is taken. A HIER data structure keeps track of the hierarchy information by using tree objects. The instances make up a hierarchical view of the design data. The top node of the tree equals the design top cell. Additional instances and connections are added according to the specification in the IO.xls and Regs.xls input files.

Various checks verify the consistency of the design hierarchy and connection matrix, e.g. detect instances not connected to the main hierarchy or drivers and loads. The check routines print out warning messages and flag the parts that conflict with the design guidelines. Based on these comments refinements to the input files should be applied by the designer. A rerun of the generator quickly shows the effect of the changes. A table-based approach for protocol specification is proposed in [5]. Even if the applications are different, both table-based approaches share the same benefits. Tables are relatively easy to read by both humans and machines and “easy” errors in tables can be caught by checks based on Boolean rules.

Ports are added to the hierarchical blocks for signals crossing hierarchies. The resulting HIER and CONN data structures represent the top levels of the chip design. The data is written to disk in an internal data format, ready to be read in by the step 2. On demand an EXCEL workbook can be generated for review and for documentation purposes.

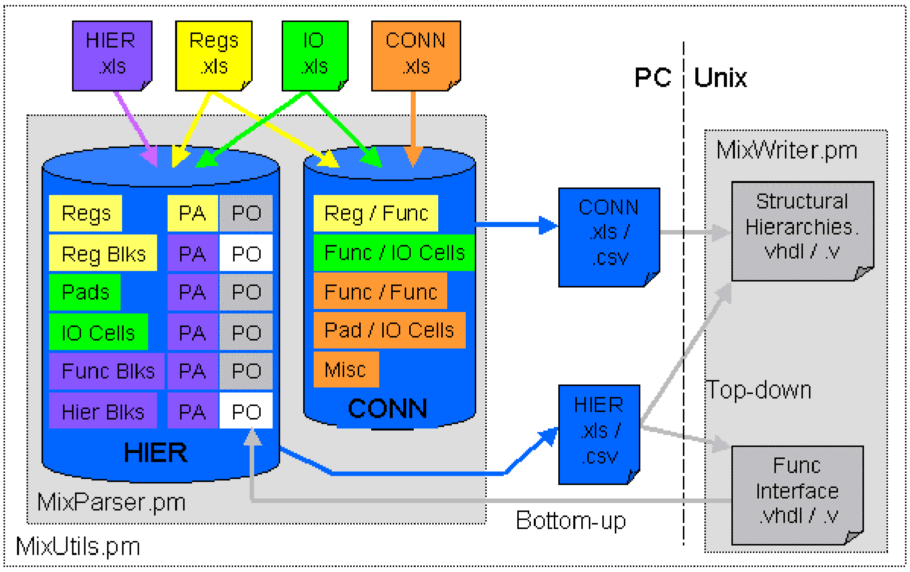

Figure 8: Dataflow from Specification to Design

4.3 Step 2: Generation of HDL Files

After transfer of the dumped intermediate data to a UNIX environment, the HDL files are generated. In case of VHDL output the primary design units Entity, Architecture and Configuration are derived and written. Additionally, HDL dependent extensions or wrappers and tool specific extensions are added and written into files.

4.4 Cross-Platform Implementation Summary

The base module MixUtils.pm provides functions for reading and writing files in various formats and for the tool configuration environment. It hides all operating system details from the upper levels, see Figure 10. MixParser.pm and MixWriter.pm correspond to step 1 and 2. Simple scripts use the functions of these modules, hiding all implementation details and data structures from the user.

5 APPLICATION RESULTS

The proposed methodology and the prototype tool was successfully applied to a display processor and scaler for LCD TV applications [6]. The design was modeled using 60 leaf modules at 7 levels of hierarchy connected with 500 explicit signals, about 250 generated register instances with 500 parameters resulting into the same number of generated implicit signals. Further, the design consisted of 90 digital pads with 4 functional and 2 test multiplex options each, resulting in around 1000 generated connections between pads IO cells and core signals. Out of 4 specification tables more than 25000 lines of VHDL code in 78 files were generated.

6 CONCLUSION

The fact than only a quarter of all connections were specified explicitly and that the rest was generated automatically, demonstrates the efficiency of the approach. But even more important than the productivity increase is the single source specification principle that guarantees consistency between all generated parts: HDL code, low-level driver software and table-based documentation.

REFERENCES

1. Sandler, S.: Expanding Debug Technology; Electronic Engineering Design - UK, Nov. 2002; http://www.electronicengineering.com/story/OEG20021105S000.
2. Edwards, C.: Mentor extends HDL portfolio with interface design; EE Times UK; 2001; http://www.electronicstimes.com/story/OEG20010309S0027.
3. Rowson, J.; Sangiovanni-Vincentelli, A.: Interface-Based Design; Design Automation Conference, 1997.
4. Schneider, C.; Ecker, W.: Stepwise Refinement of behavioral VHDL Specifications by Separation of Synchronization and Functionality; EURO-DAC with EURO-VHDL, 1996.
5. Mani Azimi, Ching-Tsun Chou, Akhilesh Kumar, Victor W. Lee, Phanindra K., Mannava, and Seungjoon Park, "Experience with Applying Formal Methods to Protocol Specification and System Architecture", Formal Methods in System Design, Vol. 22, pp. 109-116, 2003.
6. Hahn, M.; Schu, M.; Keller, S.; Schneider, C.; Carpentier, D.: System-On-Silicon for Liquid Crystal Displays; IEEE International Symposium on Consumer Electronics, ISCE, 2002.