MASTER

Development of an emulation flow as part of the PCALE design flow

van Lieshout, L.P.M.

Award date:
1994

Link to publication
Master's Thesis

Development of an emulation flow as part of the PCALE Design Flow

L.P.M. van Lieshout

Department: Department of Electrical Engineering
Design Automation Section
Eindhoven University Of Technology

Supervision: prof. dr. ing. J.A.G. Jess

Period: 01-04-1993 to 31-01-1994

Place: DVP group
Department PCALE
Philips Semiconductors

Supervision PCALE: ir. J.A.A.M. van den Hurk
ing. J. Lakerveld
ir. R.H. van der Wal

Id.nr.: 269075

The Eindhoven University of Technology is not responsible for the contents of thesis reports
Abstract

Emulation is the verification in hardware of the performance of a system prior to implementation in Application Specific Integrated Circuits (ASICs). Nowadays, the fast development of emulation systems comes within reach due to the possibility to implement a system in flexible hardware.

An emulation flow has been developed within the PCALE Design Flow. The PCALE Design Flow is a hierarchical design flow, that is used for system design. The emulation flow prescribes the consecutive steps to take to implement a design in EPLDs, starting with a VHDL description of the design.

One of the most critical steps of the emulation flow is synthesis by means of a synthesis tool. The performance of two synthesis tools has been evaluated in relation to the PCALE design style. The PCALE design style is a VHDL writing style defined for simulation purposes.

In order to increase the performance of the evaluated synthesis tools, design templates have been written to force the designer to write VHDL in a synthesizable PCALE design style. Furthermore, a Design Style Assistant tool has been developed to verify and change designs according to the templates. The use of the defined templates and the use of the developed DSA tool in the emulation flow makes emulation possible within the PCALE Design Flow.
Summary

Emulation is the verification in hardware of the performance of a system prior to implementation in Application Specific Integrated Circuits (ASICs). Nowadays, the fast development of emulation systems comes within reach due to the possibility to implement a system in flexible hardware. Emulation offers the following, attractive possibilities:

- Fast-prototyping
- Start-up production in gate arrays
- Field-test of a design
- Real-time simulation on system-level

The subject of this Master's Thesis report is the development of an emulation flow within the PCALE Design Flow. The PCALE Design Flow is a hierarchical design flow, that is used for system design. All designs at PCALE are described in a VHDL writing style. This style is called the PCALE design style and was defined for simulation purposes.

The developed emulation flow prescribes the consecutive steps to take to implement a design in EPLDs, starting with a VHDL description of the design. One of most critical steps of the emulation flow is synthesis by means of a synthesis tool. Therefore, the performance of two synthesis tools has been evaluated. This evaluation has been done in relation to the PCALE design style. This evaluation leads to the following conclusions:

- VHDL used as High Level Synthesis language offers the designer too much freedom in description style. Even VHDL written according to the PCALE style can cause problems during synthesis.
- High Level Synthesis involves various VHDL subsets, for instance the synthesis tool supported subset. Many synthesis problems stem from the fact that these VHDL subsets have non-overlapping parts.
- VHDL synthesis tools show increased performance if a design description is written on RTL level.
- It is very important to know the hardware implementation of the used VHDL constructs, to avoid complex implementations. Complex implementations decrease the clock frequency. Since, PCALE designs are used in video applications, these designs require a high clock frequency, so complex implementations must be avoided.

These conclusions lead to the following design decisions for the emulation flow. First of all, templates have been written to limit the VHDL design constructs and to guide the designer in writing synthesizable VHDL. Furthermore a Design Style Assistant (DSA) tool has been developed to verify if the description of a design complies with the defined templates. The DSA tool detects all known synthesis problems and even solves some of them automatically. The DSA tool has been verified and is correct. So the use of the defined templates and the use of the developed DSA tool in the emulation flow, makes emulation possible within the PCALE Design Flow.
However, there are still some refinements possible to the emulation flow developed so far. A good synthesis tool increases the performance of the total emulation flow. Synthesis tools are still under development so research has to be done to evaluate new developments. For the same reason, support of the templates and the DSA tool is necessary. Finally, the use of the templates by a designer could be improved, for instance by presentation of the templates in a graphical environment.
# Table of Contents

1. Introduction ................................................................. 1

2. The PCALE Design Flow .................................................. 5

3. Synthesizable VHDL ....................................................... 13

4. Evaluation of synthesis tools ............................................. 17
   4.1. Evaluation of synthesis tool CORE ......................... 19
   4.1.1. Synthesis problems ........................................... 19
   4.1.2. Timing problems .............................................. 20
   4.2. Evaluation of the synthesis tool Autologic .............. 22
   4.2.1. Synthesis problems ........................................... 22
   4.2.2. Timing problems .............................................. 27
   4.3. Conclusions with regard to synthesis tools .............. 27

5. Solution for synthesis and timing problems ..................... 29

6. Definition of templates .................................................. 33
   6.1. One specified rising edge notation ............................ 34
   6.2. Ranging of signals and variables ............................... 35
   6.3. Workaround for INOUT variables ............................ 35
   6.4. Resource sharing .................................................. 38
   6.5. Probing signals .................................................... 44
   6.6. Type conversion functions ....................................... 44
   6.7. FOR loops .......................................................... 45
   6.8. Relational operators ............................................. 45
   6.9. Record types ...................................................... 45
   6.10. Construct of a state machine .................................. 46

7. The Design Style Assistant tool ....................................... 49
   7.1. Tool control ........................................................ 49
   7.2. Actions of the tool ............................................... 51

8. The Mapping tool .......................................................... 53
   8.1. Load a design ..................................................... 53
   8.2. Map a design ..................................................... 53
   8.2.1. Logic Array Block ........................................... 54
   8.2.2. Macrocell ..................................................... 55
   8.2.3. Expander product terms .................................... 55
   8.2.4. Programmable Interconnect Array .................... 57
   8.2.5. I/O control blocks ........................................... 57
   8.3. Analyse the timing of a design ............................... 58
   8.4. Simulate a design ............................................... 58
   8.5. Program an EPLD ................................................. 58
9. Testing ........................................................................................................................ 59  
 9.1. Testcase ............................................................................................................. 59  
 9.2. Testing the DSA tool ...................................................................................... 61 

10. Conclusions and recommendations ...................................................................... 65  
 10.1. Conclusions ..................................................................................................... 65  
 10.2. Recommendations ........................................................................................... 66 

Appendix A. List of References ............................................................................. 67 
Appendix B. Brief description of QDMC IC .......................................................... 69 
Appendix C. Supported VHDL subset of synthesis tools ....................................... 71 
Appendix D. The Demultiplexer/Descrambler ..................................................... 77 
Appendix E. VHDL code of testcase State machine ........................................... 79 
Appendix F. Test results ........................................................................................... 85 
Appendix G. Test result of the DSA tool ............................................................... 87
1. Introduction

When developing new systems, it is necessary to verify their performance prior to imple­
mentation in Application Specific Integrated Circuits (ASICs). For instance, in the case of
digital video applications, simulations can be used to inspect and evaluate video images
before such a digital video application is implemented in an Integrated Circuit (IC). This
offers the possibility to critically evaluate systems prior to their implementation. In this
stage of system design changes in system specifications can still be easily made since soft­
ware can be adapted quickly, while changes in dedicated ICs (ASICs) are costly and much
more time-consuming. This strategy is incorporated in the ASIC design flow currently at
use at the Product Concept and Application Laboratory Eindhoven (PCALE). This ASIC
design flow is called the PCALE Design Flow.

The PCALE Design Flow prescribes the consecutive steps to be taken in dedicated IC
design. However, system design involves more than the development of dedicated hard­
ware only. For instance, most systems consist of both hardware and software. Also miss­
ing in the PCALE Design Flow is a flexible hardware route. This flexible hardware route
has to enable the quick development of hardware with the same functionality as the final
ASIC before ASIC design has even started. This hardware, also known as bread boards,
can then be used for emulation: a combination of the advantages of a flexible software
simulation with the advantages of real time (and consequently fast) hardware. In fact, the
reasons for integrating hardware emulation in the PCALE Design Flow are fourfold:

1. Fast-prototyping

Through emulation a customer can quickly be provided with a "prototype" of the final
ASIC (in fact emulation does not provide a prototype but a bread board with the same
functionality as the final ASIC). The availability of a prototype enables the customer to
verify his specification through testing the functionality of the bread board. This allows
tracing desirable changes in the specification in an early stage of the ASIC design. Fur­
thermore, the customer can start writing software for his application (in case software is
part of the system) and build a prototype-system. In general, a total of some tens of pro­
totype copies can be expected since prototypes are usually small in number.

2. Production

A second possibility is to map a description of the design to gate arrays in order to use
these gate arrays in the beginning of system production. Gate arrays are half-fabricated
ICs: the logic cells are already fabricated but the interconnections (wiring) still have to
be made through two final IC masks.

The use of gate arrays in start-up production is faster and less expensive and therefore
more desirable than fabricating a dedicated IC. In this case less than a 100,000 gate
array copies can be expected. Later on, an optimal and more expensive dedicated IC
can be designed for mass production.

3. Field-test

The prototype can be used for a so-called field-test. This means that incomplete parts of
the specification can be tested by the designer and that some parts can be evaluated with
respect to their functionality. The incomplete parts of the specification can usually be completed after such a field-test.

4. Real-time simulation

Through emulation, the designer has the opportunity for real-time simulation. This way "simulations" (by means of emulations) can be carried out much faster than traditional simulations. In particular for simulations at system-level, a large reduction in simulation time is to be expected. Emulation does not mean that simulation has become outdated: through simulation a description of a design must be checked for correctness; after that, by emulation, the design becomes rapidly available in hardware without having to wait until the ASIC design has been completed.

Yet building bread boards in the usual way is time-consuming and not very flexible. Fortunately the quick development of bread boards comes within reach due to the emergence of flexible hardware modules. But the bread board development speed is not the only requirement that is imposed on a flexible hardware route (emulation flow).

Another requirement is that the emulation flow starts with a description of a design in a Hardware Description Language (HDL), a language especially developed and suited for the description of hardware designs. Several of such HDLs exist, but the HDL that is used for this purpose at PCALE is the VHSIC Hardware Description Language (VHDL). This HDL is defined by the Institute of Electrical and Electronics Engineers (IEEE) and is used in the industry for the description of designs during development (see [2]). This requirement is imposed on the emulation flow, since the basis of the dedicated hardware route, an HDL design description, must be the basis of the flexible hardware route also, in order to ensure identical functional behaviour of the ASIC and bread board.

Furthermore the emulation flow must fit into the PCALE Design Flow. This means that the mandatory functional verification at all levels of the PCALE Design Flow must also be applicable to the levels of the emulation flow.

Fourth requirement on the emulation flow is that the application of the emulation flow has to be kept in mind: the emulation flow is to be used for designs that involve video applications, so very stringent speed requirements have to be taken into account.

Finally, a choice has to be made what flexible hardware modules to use. There are several choices for flexible hardware modules since a number of such devices are available on the market: gate arrays from different vendors (Altera, Xilinx, Actel, etcetera) and Erasable Programmable Logic Devices (EPLDs) from Altera. The Digital Video Processing (DVP) group at PCALE has chosen to use EPLDs from Altera as their flexible hardware modules for several reasons:

1. Only for large amounts of bread boards (for instance when emulation is to be applied for production start), gate arrays are cheaper than EPLDs. Since the first applications of the emulation flow apply to fast-prototyping and field-testing (hence a small amount of bread boards), EPLDs are considered as back end of the emulation flow.
2. EPLDs are reprogrammable while gate arrays can only be given a certain logic function once. With EPLDs as flexible hardware, this flexible hardware is re-usable when a bread board is no longer needed. But the fact that EPLDs can be quickly reprogrammed has an additional advantage. As with all developments, the emulation flow too has to be tested several times during its development. Using gate arrays for such tests is too expensive and takes too much time. EPLDs on the other hand can be used for several tests and their programming takes little time. On top of that, the EPLDs can even be used for a bread board after the emulation flow has been developed: when testing the emulation flow with EPLDs, no money is lost on flexible hardware. Of course, after the emulation flow has been developed, the extension to gate arrays can then still be made.

3. Altera EPLDs are the fastest devices according to comparisons with other flexible hardware modules. These comparisons are based on benchmarks (well-known and well-defined designs used as standard testcase) and have been performed by the Programmable Electronic Performance Corporation (PREP), a consortium of 13 prominent suppliers of programmable logic and tools.

4. Altera EPLDs have been used before by the DVP group. Very satisfactory performance was experienced on those occasions. So there is no reason for changing to new and unknown devices unless they prove to be better.

The next chapter discusses the PCALE Design Flow in its present form (the Existing PCALE Design Flow) and in its successor form (the Advanced PCALE Design Flow), followed by an elaboration of the emulation flow. Chapter 3 concerns all the aspects of VHDL that are involved in the development of the emulation flow. Chapters 4 to 8 involve the development of the emulation flow itself. In chapter 9, a testcase of the emulation flow is explained. The final chapter is concerned with conclusions and recommendations regarding the development of the emulation flow.
2. The PCALE Design Flow

As already stated in the introduction, the PCALE Design Flow in its present form does not capture all elements of system design. The DVP group has set out to extend this design flow to a design flow that covers more and hopefully all aspects involved in system design. One of the extensions is the emulation of designs, including the emulation of designs with large memory requirements. However, before the emulation flow for designs with large memory requirements is developed, it is important to have a good notion of the PCALE Design Flow in its present and in its envisioned form and of the standard emulation flow developed for designs in general. They are discussed in this chapter.

2.1. Existing PCALE Design Flow

The PCALE Design Flow, depicted in figure 1 on page 6, is a top-down hierarchical design flow. It prescribes a trajectory from algorithm to evaluated silicon and is based on two basic principles: specification and verification. As for the first principle, the paper specification of a design is the input for the design flow and must be very accurate since the functionality of the flow input highly determines the functionality of the flow output, the final ASIC. The second basic principle, the functional verification at all levels of the flow, is to ensure design correctness at every moment during design development including the flow output. The combination of the two basic principles is the philosophy behind the PCALE Design Flow, which yields a lot of advantages over non-hierarchical design flows. The most important advantages are:

- A reduced risk of functional design errors
  
  This is the most important benefit of mandatory functional verification at all levels in the flow.

- An integrated design environment for system development
  
  This allows for straightforward data exchange between tool sets and between consecutive design levels.

- A short throughput time
  
  A direct result of a short throughput time is a short Time-To-Market.

- The possibility to join forces of multiple design teams in the development of a chip-set

- The possibility to limit simulation run times
  
  Through abstract functional descriptions of individual ICs system behaviour is matched with the algorithm specification and simulation at high abstraction levels becomes possible, resulting in limited simulation run times.

The PCALE Design Flow has been successfully applied during the development of the first generation of HD-MAC Bandwidth Restoration Decoders (BRDs) in the Eureka-95 project, which involved the development of High Definition Television (HDTV). It proved to be very effective and is now being used for digital design at PCALE. For a more extensive description on this design flow, see [1].
The PCALE Design Flow starts at the Algorithm Level (AL). In this stage of the design flow a system’s functional behaviour is recorded in an abstract software description. This description is known as the reference software, or algorithm. An algorithm is the principal functional reference for the development of a system in the PCALE Design Flow.

Once an algorithm has been frozen, IC-partitioning is performed. For each IC in an IC-partitioning, its behaviour is described in a High Level (HL) description. An HL is used as functional reference for the development of an individual IC. IC interfaces and functional behaviour must be in exact accordance with the HL. The combined behaviours of all HLs must be equivalent to the algorithm’s behaviour.

To capture an IC’s proposed interior architecture and hierarchy, a Medium Level (ML) description can be written which is less abstract than an HL. Functional correctness of an ML is verified through bit-by-bit comparison with the HL description; bit-by-bit comparison is performed through simulations. An ML is written in a Hardware Description Language (HDL) at Register Transfer Level (RTL).

The lowest level symbolic description of an IC, the Library Level (LL) description, is created by implementing the ML by means of library elements. Such an LL contains both symbolic representations of VLSI library blocks and their symbolic interconnections. Functional correctness of an LL is verified through bit-by-bit comparison with (parts of) the ML. Timing verification is performed also.

Through placement and routing, the IC layout is generated from an LL description. This layout is checked during factory finishing, for instance to find possible design rule errors.

In this stage of the PCALE Design Flow, the IC layout is transferred to a foundry. At the foundry the design is implemented on silicon wafers and the first IC prototypes are delivered to the design team for testing.

When the first IC prototypes return from the foundry, silicon evaluation can start. Silicon evaluation includes both functional and electrical evaluation. In addition to IC-only evaluation, (sub)system evaluation, including other ICs in the chip-set, is performed.

**FIGURE 1. Existing PCALE Design Flow**
2.2. Advanced PCALE Design Flow

Until now system development was separated into the development of the hardware part of the system, followed by the development of the software part (provided that the system incorporates both hardware and software); on top of that the two developments were cast in a different mould. The PCALE Design Flow in its present form as described in the preceding section, prescribes the consecutive design steps to take in dedicated hardware design. However, due to a growing understanding of system design and all the aspects of system design over the years, the idea was formed that a complete design flow should cover all the aspects of designing and not merely dedicated hardware design. Hence, the DVP group at PCALE set out to extend the Existing PCALE Design Flow. The PCALE Design Flow in its envisioned extended form, called the Advanced PCALE Design Flow, is shown in figure 2.

![Advanced PCALE Design Flow Diagram](image)

**FIGURE 2. Advanced PCALE Design Flow**
It is important to realize that the philosophy behind the Existing PCALE Design Flow remains intact in the Advanced PCALE Design Flow. The difference is that this philosophy is applied to other aspects of system design also (for example to the development of a system's software). Another important notice is that the Existing PCALE Design Flow in figure 1 is really a design flow in the sense that it identifies the various levels and the consecutive steps involved in dedicated hardware design. The diagram of the Advanced PCALE Design Flow in figure 2 is conceptually different since it merely identifies possible target implementations. The concept of different levels during system development still applies although these levels are not depicted in figure 2.

The blocks in the Advanced PCALE Design Flow are:

- **Paper specification**

  Completely analogue to the Existing PCALE Design Flow, the Advanced PCALE Design Flow starts with the paper specification of the system. Based on this specification, the system is developed. A system can consist of both hardware and software. Instead of separating the development of a system's hardware and software in two consecutive and conceptual different steps, the co-design of the two has a lot of advantages, namely:

  1. The hardware-software combination can be tested in an early stage of system development. This in turn offers the possibility to check the specification of the complete system at an early hour against customer wishes. This system evaluation can then be used to adjust or complete the specification. Most likely this leads to better designs and largely reduces the possibility of redesigns.

  2. Furthermore, simultaneous hardware and software design decreases the total Time-To-Market. The total Time-To-Market is IC development time plus software development time. The Time-To-Market (IC development time) in the Existing PCALE Design Flow is already much shorter than the Time-To-Market of non-hierarchical design flows. However, if a system also incorporates software, then the software development time is not accounted for in this Time-To-Market. The total Time-To-Market decreases due to the co-design of hardware and software in the Advanced PCALE Design Flow.

  3. At some stage in system design, the system has to be partitioned in hardware and software. With a growing knowledge of the system during development, this partitioning can be adjusted on the basis of an estimation of costs. This estimation can be thought of as a function taking into account Customer requirements, Overall development cost, Silicon area & package and Time-To-Market (COST). The partitioning adjustment can be made in almost every stage of the design flow since both hardware and software are described in the same description language, for example VHDL.

  The combination of hardware and software development, Hardware-Software Co-design, is therefore captured in the Advanced PCALE Design Flow, starting with the paper specification.
It is important to observe that both hardware and software are based on VHDL descriptions. VHDL was developed for the description of hardware as the name already suggests, VHSIC Hardware Description Language. But taking a closer look at VHDL, it is observed that it incorporates certain constructs that can be used for the description of software also. This in fact makes the smooth hardware-software integration feasible and worthwhile; otherwise Hardware-Software Co-design becomes much more complex and perhaps not even feasible within the PCALE Design Flow.

- **Functional model**
  Also completely analogue to the Existing PCALE Design Flow is recording a system’s behaviour in an abstract software description. Again this description (or algorithm) is the principal functional reference for the development of a system. The only difference with the Existing PCALE Design Flow is that a system involves both system hardware and system software in the Advanced PCALE Design Flow. Therefore the algorithm incorporates the combined functionality of a system’s hardware and software.

- **Software**
  Based on the evaluation of the COST function, some parts of the system are selected to be implemented in software. A distinction can be made between firmware and microcontroller (μC) software. Firmware is fixed software, which means that this software possesses little or no flexibility (for instance software in a ROM). Software implemented on a micro-controller is much more flexible, but on the other hand takes more chip area. For some designs firmware suffices while other designs need the micro-controller implementation; sometimes even, the designer has to evaluate the pros and cons of the two before making a choice.

- **Micro-controller (μC) software**
  Just like all target implementations, the final implementation in micro-controller software is based on a VHDL description. But VHDL is not suited for programming a micro-controller. Hence a translation from VHDL to some programming language is necessary for implementation in a micro-controller: a tool translating sequential VHDL to the C programming language has already been developed at PCALE.

- **Firmware**
  Firmware, being the fixed implementation of software, is already indicated as part of the Advanced PCALE Design Flow. Yet the design flow for firmware is still to be developed.

- **Hardware**
  Based on the evaluation of the COST function, some parts of the system are selected to be implemented in hardware. Final implementation usually means development of dedicated hardware (implementation in ASICS). However, besides dedicated hardware also the implementation in flexible hardware is possible. This implementation is usually of a more temporary nature since it is used for emulation purposes.
The fact that both dedicated as well as flexible implementations can be derived for hardware, has an additional advantage. It is possible to implement ICs from a chip-set via the dedicated route one by one. The others can be emulated until an IC has been implemented in dedicated hardware. Then another IC follows the dedicated path until the whole chip-set is available in dedicated hardware.

- **Flexible hardware**
  As mentioned in the introduction, emulation can be very useful. The block called flexible hardware indicates the route that leads to emulation boards (bread boards). A flexible hardware route for designs is subject of this Master's Thesis.

- **Dedicated hardware**
  Dedicated hardware is the development of ASICs: selecting the dedicated path for parts of the system means that these parts are implemented in ASICs. The path Paper specification - Functional model - Hardware - Dedicated in the Advanced PCALE Design Flow indicates the target technologies of the Existing PCALE Design Flow. This means that all dedicated hardware is developed according to the Existing PCALE Design Flow. For dedicated hardware, two blocks in the Advanced PCALE Design Flow are distinguished: full-custom and synthesis.

- **Full-custom**
  Dedicated full-custom hardware design means development of ASICs that are as optimal as can be. The design team exerts itself to the utmost to optimize the final ASICs. The consecutive steps to take in full-custom hardware design are prescribed by the Existing PCALE Design Flow.

- **Synthesis**
  The block synthesis in the Advanced PCALE Design Flow indicates the development of all dedicated hardware except full-custom hardware design. Final target implementations are standard cell or datapath designs. The derivation of these implementations is prescribed by the Existing PCALE Design Flow.

### 2.3. Standard emulation flow

Now that the reasons for emulation and the place of emulation in the PCALE Design Flow have been determined, it is time to take a closer look at the emulation flow itself. As mentioned in section 2.2, emulation is the implementation of an HL description of an IC in flexible hardware. This implementation is of use when the final ASIC has not yet been developed through dedicated design. Only by using synthesis tools the flexible hardware implementation can be generated *quickly* (which is an essential demand on the emulation flow). After synthesis, a mapping has to be generated by a mapping tool and finally the generated mapping can be transferred to flexible hardware. These are the main steps in the emulation flow.

However, emulation is to be part of the PCALE Design Flow and must therefore comply with the philosophy behind the PCALE Design Flow. So every next step in the emulation flow can only be taken if the functional correctness of the preceding step has been estab-
lished. In figure 3 the concept emulation flow is illustrated along with the three bit-by-bit comparisons that have to be performed to verify functional correctness. The first two are based on simulation results; establishment of functional correctness of the programmed flexible hardware is in fact bit-by-bit comparison of simulation results with emulation results.

As mentioned before in the introduction, VHDL is used as HDL. The next chapter concerns all the aspects of VHDL that are involved in the development of the emulation flow.
3. Synthesizable VHDL

VHDL is being introduced in VLSI design, and is the High Level input of the emulation flow mentioned in the previous chapter. VHDL is the HDL that is used at PCALE for design descriptions. High-Level Synthesis has received much attention lately, mainly because it helps to cope with the ever growing complexity of digital systems. VHDL is a formal notation for hardware description that can be used to create a HL description of digital systems. Using VHDL for High-Level Synthesis combines the advantages of a standard with the advantages inherent in High-Level Synthesis. In 1987, VHDL was accepted as standard High Level Hardware Description Language (IEEE 1076-standard, see [2]), but since 1991 it is also used as High Level Synthesis language (see [15] and [16]).

Although VHDL supports the development, verification, synthesis and testing of hardware designs there are some difficulties for synthesis. One of these difficulties is that VHDL contains statements, which have no hardware implementation. For instance, it is possible to describe a "read file" action in VHDL. Statements used to describe such a "read-file" action cannot be synthesized.

Statements without a hardware implementation signify a fundamental gap between a standard VHDL description and a synthesizable VHDL description. But it is possible to use a slightly restricted standard VHDL as the source for both High-Level Synthesis and simulation, see [15].

In figure 4 a division of VHDL in two subsets is shown:
1. A subset that contains VHDL that is simulatable but not synthesizable.
2. A subset that contains VHDL that is both simulatable and synthesizable.

VHDL according to IEEE 1076-standard
Simulatable VHDL

Only simulatable, unsynthesizable VHDL
Simulatable and synthesizable VHDL

FIGURE 4. Simulatable and synthesizable VHDL subsets

VHDL is getting accepted in the Philips IC design community and is currently being used as a specification language for applications, as well as an architecture description language. In order to get a smooth integration and environment description, a document has been written containing guidelines for modelling digital electronic hardware designs in
VHDL, see [3]. These guidelines define another subset of VHDL, called the PCALE subset. Some statements of standard VHDL are restricted in this subset because use of them increases simulation times. For instance, signals must be used as least as possible because many signals increase simulation time.

This subset contains both simulatable VHDL statements and VHDL statements that are synthesizable and simulatable. In figure 5, the relation between the PCALE subset and the standard VHDL subset is shown.

Besides all these restrictions, it is still necessary to make another division. In the emulation flow a synthesis tool is used and the VHDL that is supported by a synthesis tool is not the same as the VHDL written according to the PCALE subset. The tool only supports a large part of synthesizable VHDL and ignores constructs that do not have hardware implementations, for instance reading data from a file. Furthermore, the development of synthesis tools was started about 1991, so nowadays synthesis tools are still under development and do not support all synthesizable VHDL, yet. This implies another subset in the ellipse, the tool supported VHDL subset, see figure 6. This subset differs from synthesis tool to synthesis tool; but since in practice only one tool is used, only one (abstract) subset is depicted.
The tool supported VHDL subset does not support all constructions of the PCALE subset. Therefore, VHDL written according to the intersection of the tool supported subset and the PCALE subset, is used as the input of the emulation flow. A problem is that the style of writing in this subset is synthesis tool dependent, but with this style of writing it is possible to describe and simulate a design as well as to use the description as input for a synthesis tool. It is desirable if the overlap between the tool dependent subset and the PCALE subset is maximal. Two synthesis tools are evaluated to determine if writing in this style guarantees a synthesizable description. The next chapter discusses this evaluation.
4. Evaluation of synthesis tools

It is very important to select a proper synthesis tool because the style of writing VHDL depends on it. So in order to test the performance of synthesis tools, several tests have been done.

Two tools are evaluated each with its own VHDL entry level. These tools can synthesize and optimize a design with EPLDs as target implementation. The tools are:

- Autologic from Mentor Graphics.
- CORE from Exemplar.

Each tool has its own strategy for synthesis and optimization, and its own VHDL style. Therefore an evaluation has been done to select the tool that best fits into the PCALE Design Flow. A testcase has been written according to the PCALE style. However, synthesis tools have their own tool supported subset, as described in the previous chapter. Besides differences in subsets, each synthesis tool prefers its own synthesis notation style. One goal of the evaluation was to find all differences in notation style between the PCALE style and the style preferred by CORE or Autologic.

The testcase, used to evaluate both tools, describes a state machine. This state machine has been written to both the CORE supported subset and the Autologic supported subset. Preferably, one testcase is used for both tools. However, due to differences in tool supported subsets, the testcase is slightly adapted for each synthesis tool. Yet the functional behavior of the testcase is not changed by the adaptations. So comparison of the synthesis tools is valid.

Another goal was to find the best description to obtain an optimal result with CORE or Autologic. All designs have to function correctly on a relative high clock frequency (about 27 MHz), because applications for video are developed.

The strategy used to evaluate and compare both synthesis tools is depicted in figure 7. Prior to the evaluation of the tools, correctness of the functional behavior of the state machine was established by simulation with a VHDL simulator (VANTAGE). This way the evaluation is based on correct (valid) VHDL code. For the evaluation of the tools itself, the functional behavior of the testcase is not relevant. The interested reader is referred to appendix D for a detailed description of this testcase. Also, a more extensive testcase has been used to test the complete emulation flow. This testcase is discussed in chapter 9. The conclusions of this evaluation are discussed in section 4.3.
Independent of the evaluated synthesis tools, two global problems have been found and these problems are discussed first:

- A design often contains memory. This memory can be described by large std_logic_vectors in VHDL. In the next example a segment register is used to store data. This data can be read in other processes.

```vhdl
VARIABLE segment_register : std_logic_vector(1839 DOWNTO 0);
VARIABLE write_pointer : integer RANGE 0 TO 1839;
...
-- data from a data register is stored in segment register
segment_register(write_pointer+7 DOWNTO write_pointer) := data_register;
... -- other processes can access data of segment register
```

Due to the fact that flexible hardware elements have very little memory capacity and the fact that large registers require a large memory capacity to be available in the flexible hardware, synthesis tools have to use a lot of flexible hardware elements (EPLDs) when mapping a design containing large registers to flexible hardware. This in turn means that the design has to be partitioned among multiple EPLDs. Partitioning implies
an increase in wiring and in wiring complexity which leads to inefficient mappings which in turn cause a decrease in clock frequency. Also it is preferred to keep the number of flexible hardware elements as small as possible in order to keep the bread board simple, small and as cheap as possible.

Besides many flexible hardware elements, a synthesis tool crashes during synthesis of such designs, probably because such large registers cause an overflow in the internal format used by the synthesis tools. To overcome this problem, the large register could be replaced by a RAM under certain constraints. This solution is discussed in the Master’s Thesis of K.J. Lammers, see [19], and requires changes in the original VHDL code.

- Internal signals can be probed by including them in the port map of an entity. If an internal signal inside a process is assigned to a signal of the port map of the entity then the synthesis tool generates a flip-flop to latch the internal signal and a flip-flop to latch the port map signal. This implies extra delay and causes timing problems.

4.1. Evaluation of synthesis tool CORE

During synthesis, two kinds of problems were found. First, there were problems writing a design according to the CORE supported VHDL subset (see [11]) and according to the PCALE subset. In appendix C, the subset of VHDL supported by CORE is shown. Besides the restrictions of this subset, the designer must follow the guidelines of CORE for the style described in the manual, see [12], otherwise the tool cannot always recognize all VHDL constructs. Some important requirements of this style are mentioned in the next section.

The design was mapped to an EPLD after synthesis of the state machine. This EPLD mapping was simulated to verify functional correctness and to verify if the simulation results are identical with the first simulation before synthesis. Simulating this design, it turned out that there were timing problems, as is explained in section 4.1.2.

4.1.1. Synthesis problems

The VHDL constructs that can cause problems during synthesis with the synthesis tool CORE are listed below:

- Synchronous designs often use a rising edge to synchronize the design. Each synthesis tool prefers its own rising edge definition. The rising edge definition recommended by CORE is VHDL code of the following form:

  IF (clock = '1' AND clock'EVENT) THEN

  or:

  WAIT UNTIL (clk'EVENT AND clk = '1');
There are many other definitions for defining a rising edge, but these statements are not recognized as a rising edge by CORE. So the tool builds complex logic to implement other rising edge definitions.

- Procedures are not synthesized. The complete procedure body has to be included into the main program in order to synthesize correctly.

- It is possible to index parts of std_(u)logic_vectors. These parts are called slices. One must take into account that the slice boundaries of a std_(u)logic_vector must evaluate to integer constants. So the following statement generates an error because this construct cannot be synthesized:

```
VARIABLE pkt_reg: std_ulogic_vector(31 DOWNTO 0);
VARIABLE id_len : integer RANGE 0 TO 31;
...
pkt_reg(id_len-1 DOWNTO 8) := pkt_data_reg(id_len-9 DOWNTO 0);
```

In this case another description has to be found by the designer.

- CORE does not support "Z" assignments in processes or complex expressions. So to describe the communication with a bidirectional bus, one must place all "Z" assignments outside the process body without changing the functional behavior.

- The use of short names of identifiers is recommended. Otherwise, the names are abbreviated by the EPLD mapping tool and these names become unreadable.

The VHDL code describing the state machine has been changed according to all problems mentioned above. After a successful run, the design was mapped to an EPLD and this EPLD mapping was simulated again to verify functional behavior. The results of this simulation are discussed in the next section.

### 4.1.2. Timing problems

The simulation results of the EPLD mapping obtained after synthesis, were correct but only at low clock frequencies, lower than 6 MHz. As mentioned before, the minimum frequency for video processing is 27 MHz. To meet the 27 MHz goal, several tests have been carried out to find constructs and options to improve the timing. It turns out that a designer must be aware of the hardware consequences of the VHDL code that has been written. If constructs are used that lead to complex implementations, then this affects the timing negatively. Therefore some global recommendations are given that influence the final results in a positive manner:

- CORE does not perform resource sharing. The next example describes a part of the state machine in the test case. A counter, named counted_bits, counts the bits of the input. Depending on the value of counted_bits, the next state is determined. If the following description is used, the tool implements two counters:

```vhd
CASE nextstate IS
  WHEN state_1 =>
    counted_bits := counted_bits + 1;
```

Development of an emulation flow
nextstate := state_2;

WHEN state_2 =>
    counted_bits := counted_bits + 2;
    IF condition THEN nextstate := state_3;
    ELSE nextstate := state_2;
END IF;

END CASE;

Rewriting the code as follows, increases the performance:

CASE nextstate IS
    WHEN state_1 =>
        update_count:=1;
        nextstate := state_2;
    WHEN state_2 =>
        update_count:=2;
        IF condition THEN nextstate := state_3;
        ELSE nextstate := state_2;
END IF;
    ...
END CASE;

counted_bits := counted_bits + update_count;

When synthesizing the code above, the tool implements only one counter for counted_bits. This notation (with a variable) forces the tool to share the counter (the resource). This means that resource sharing has to be done by hand, to improve the synthesis results of the design.

- All integers and vectors (bit_vectors, std_logic_vectors and std_ulogic_vectors) must be ranged; this to prevent CORE from taking the default value of 32 bits for every integer and vector.

- If possible, it is always better to replace constructs with relational operators (<,>,<=,>=) by constructs with an equal (=) comparison, for instance:

  -- counted_bits counts between 0 and 255
  IF counted_bits < 255
    THEN A;
  ELSE B;
  END IF;

  has to be replaced by:
February 3, 1994

Evaluation of synthesis tools

-- counted_bits counts between 0 and 255
IF counted_bits = 255
    THEN B;
    ELSE A;
ENDIF;

Counted_bits always has a value between 0 and 255. This adaptation only functions correctly if the start-up conditions are well defined, otherwise setup errors will occur. During simulation a lot of these setup errors can be detected. The replacement, mentioned above, results in comparators which are less complex because the comparator only has to check if counted_bits equals 255 instead of checking if counted_bits has a value below 255. This replacement decreases the gatecount of the implementation.

- FOR loops are not always synthesized efficiently or even correctly. Therefore it is better to avoid the use of FOR loops in VHDL code. The tool generates complex logic to implement the functional behavior of a FOR loop.

4.2. Evaluation of the synthesis tool Autologic

Some other problems have been found evaluating Autologic. The same testcase and the same strategy as mentioned in the introduction of this chapter, have been used. Besides the state machine, also a practical design QDMC, was used as testcase. This QDMC design has also been written according to the PCALE style. Just like testing the state machine, this design was simulated before synthesis and simulated after mapping to an EPLD.

Some adaptations have been made to write the design according to the synthesis rules of Autologic, see [9] and [10], but the functional behavior remains identical. The next section lists synthesis problems of Autologic. Problems related to the timing are discussed in section 4.2.2.

4.2.1. Synthesis problems

The VHDL constructs that can cause problems during synthesis with the synthesis tool Autologic are listed below:

- Type conversion functions in VHDL code. In the PCALE subset, integers and std_(u)logic_vectors are types that can be used in a description. Integers and std_(u)logic_vectors have the same hardware implementation, a databus of a certain number of bits (depending on the range of the integer or std_(u)logic_vector). Sometimes a conversion is needed between the two types. In VHDL, this is realized by means of type conversion functions. Therefore, the hardware implementation of a type conversion function should be a wire.

The following code shows a type conversion function written to convert an integer, into a std_ulogic_vector:

```vhdl
SUBTYPE std_ulogic_vector2 IS std_ulogic_vector(0 TO 1);
FUNCTION inttologic2 (i : integer RANGE 0 TO 3) RETURN std_ulogic_vector_2 IS
```

Development of an emulation flow
VARIABLE result : std_ulogic_vector_2;
BEGIN
CASE i IS
  WHEN 0 => result := "00";
  WHEN 1 => result := "01";
  WHEN 2 => result := "10";
  WHEN 3 => result := "11";
  WHEN OTHERS => result := "00";
END CASE;
RETURN result;
END inttologic2;

Since integers and std_(u)logic_vectors have the same hardware implementation (data­bus), type conversion functions can be implemented as wires. However, the type con­version function, described above, has been synthesized by Autologic and Autologic generates the following implementation, see figure 8.

This implementation is unacceptable because it requires 14 gates instead of a wire. It is very easy to overcome this problem because Autologic predefines such a type conversion function. Synthesis of this function results in wires instead of 14 gates. The VHDL code of the Autologic type conversion function is also available and can be used for simulation. Besides the conversion of an integer to a vector, there are many other con­versions, and the same problems occur if the designer uses his own type conversion functions. Although not explicitly stated in the previous section, CORE also has pre­defined type conversion functions for the same reason.

The recommendation to overcome this kind of problems is to use the predefined type conversion functions of the synthesis tool.

- INOUT variables in procedures. The synthesis tool, Autologic, has problems to synthe­size procedures with INOUT variables. More in particular, Autologic cannot correctly synthesize procedures with INOUT variables that imply memory. If the variables do
not imply memory, synthesis is correct. Procedures are often used to define hierarchy in a design. According to most synthesis tools, subprograms can only be used to describe combinational logic, but using INOUT variables it is possible to describe static memory in a procedure. Procedures with INOUT variables that define static memory, cannot be synthesized correctly by the applied synthesis tools.

Testing the synthesis tool, one conclusion could be made complexity of the implementation of the VHDL code strongly depends on the writing style of the description of the design. A test that illustrates this conclusion is discussed now:

- One block in the QDMC design contains a ROM. This ROM was used for the lock process, see appendix B. The ROM was specified in programming language C; a small part of this specification is stated below:

```c
static int lock_rom[8][4] = {
    {8, 8, 8, 7, 7, 5, 1, 1},
    {8, 8, 8, 8, 7, 6, 3, 1},
    {8, 8, 9, 9, 9, 7, 4},
    {7, 8, 9, 10, 12, 13, 13, 11};
```

```c
int i, q, rom_out;
...
rom_out = lock_rom[i][q];
```

In VHDL, a ROM can be described in many ways:

1. By means of an two-dimensional array. A two-dimensional array of type std_ulogic_vector has been defined. This array has to be initialized once. This description is almost identical to the C description. The VHDL code is:

```vhdl
TYPE rom IS ARRAY (0 TO 7, 0 TO 3) OF std_ulogic_vector(0 TO 3);
VARIABLE i : integer RANGE 0 TO 7;
VARIABLE q : integer RANGE 0 TO 3;
VARIABLE rom_out : std_ulogic_vector(0 TO 3);
VARIABLE lck_rom : rom;
BEGIN
lck_rom := (("0001","0001","0001","1110","1110","1010","1000","1000"),
("0001","0001","0001","0001","1110","0110","1100","1000"),
("0001","0001","1001","1001","1001","1001","1110","0010"),
("1110","0001","1001","0101","0011","1011","1011","1101"));
rom_out := lck_rom(i, q);
```

2. Combination of CASE statement and one-dimensional array. A one-dimensional array of type std_ulogic_vector has been declared and with a CASE statement the second dimension is selected.

The following code shows how to describe the ROM with a CASE statement:

```vhdl
TYPE rom IS ARRAY (0 TO 8) OF std_ulogic_vector(0 TO 3);
VARIABLE i : integer RANGE 0 TO 7;
```
VARIABLE q : integer RANGE 0 TO 3;
VARIABLE lck_rom : rom;
VARIABLE rom_out : std_ulogic_vector(0 TO 3);

BEGIN

CASE q IS
  WHEN 0 => lck_rom:=("0001","0001","0001","1110","1110","1010","1000","1000");
  WHEN 1 => lck_rom:=("0001","0001","0001","0110","1000","1000");
  WHEN 2 => lck_rom:=("0001","0001","1001","1001","1001","1001","1110","0010");
  WHEN 3 => lck_rom:=("1110","0001","1001","0101","0011","1011","1011","1101");
  WHEN OTHERS =>
    lck_rom:=("0000","0000","0000","0000","0000","0000","0000","0000");
END CASE;
rom_out := lck_rom(i);

3. As a large, one-dimensional array. Instead of two dimensions, one dimension is used, so the array is of size 1 to 32:

TYPE rom IS ARRAY (0 TO 31) OF std_ulogic_vector(0 TO 3);
VARIABLE i, q : integer RANGE 0 TO 7;
VARIABLE lck_rom : rom;
VARIABLE rom_out : std_ulogic_vector(0 TO 3);

BEGIN

  lck := ("0001","0001","0001","1110","1110","1010","1000","1000",
  "0001","0001","0001","1110","0110","1100","1000",
  "0001","0001","1001","1001","1001","1001","1110","0010",
  "1110","0001","1001","0101","0011","1011","1011","1101");
rom_out := lck(i + 8 * q);

In this case the indexing requires a multiplier (index = i + 8 * q).

4. The output of the ROM can be determined by using the address as selection, address is the sum of i and 8 * q. Again a multiplier is required.

VARIABLE i, q : integer RANGE 0 TO 7;
VARIABLE address : integer RANGE 0 TO 32;
VARIABLE rom_out : std_ulogic_vector(0 TO 3);

BEGIN
  address := i + 8 * q;
  IF ( address = 0 OR address = 1 OR address = 2 OR address = 8 OR address = 9 OR
      address = 10 OR address = 11 OR address = 16 OR address = 17 OR address = 25 )
    THEN rom_out:="0001";
ELSIF (address = 3 OR address = 4 OR address = 12 OR address = 22 OR address = 24) THEN rom_out:="1110";
ELSIF .... -- etc.
ENDIF;

5. The output of each separated bit of the ROM can be determined by using the address in a selection statement. The following code illustrates how this construct has to be coded for the ROM example, mentioned above:

VARIABLE i , q : integer RANGE 0 TO 7;
VARIABLE address : integer RANGE 0 TO 32;
VARIABLE rom_out: std_ulogic_vector(0 TO 3);
BEGIN
address := i + 8 * q;
IF ( address = 0 OR address = 1 OR address = 2 OR address = 8 OR address = 9 OR address = 10 OR address = 11 OR address = 16 OR address = 17 OR address = 18 OR address = 19 OR address = 20 OR address = 21 OR address = 25 OR address = 26 OR address = 27 OR address = 28 OR address = 29 OR address = 30 OR address = 31) THEN rom_out(3):='1';
ELSE rom_out(3):='0';
ENDIF;
... -- etc.

Description 1 and the C description are almost identical, but unfortunately, multiple arrays are not supported yet, so this description cannot be synthesized. Description 2 is a concession to this restriction, but is still a High Level description, as contrasted with the VHDL code alternatives 4 and 5 which are Register Transfer Level descriptions (RTL descriptions). RTL descriptions are not desirable because they depend too much on the implementation. HL descriptions are much more flexible and do not require knowledge of the final implementation.

Descriptions 1 to 5 have been synthesized in Autologic. The results are stated in Table 1.

<table>
<thead>
<tr>
<th>Description</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>-</td>
<td>506</td>
<td>1320</td>
<td>327</td>
<td>348</td>
</tr>
</tbody>
</table>

This table shows that using description 4 requires the least number of gates. This can be explained by the fact that the descriptions on RTL level are synthesized much better by the tool than High Level descriptions. Synthesis of High Level descriptions takes complex algorithms and synthesis is still subject of research.
So, if a designer wants to synthesize a design description, the best results can be obtained if a design is written on a RTL level. However, it is still desirable to start with a High Level description, but it seems necessary to make an exception to describe complex algorithms. Complex algorithms with a complex hardware implementation must be described on RTL level to obtain best synthesis results.

### 4.2.2. Timing problems

During the tests with the state machine, several different descriptions were used and verified. Besides the consequences for synthesis, these descriptions appear to affect the timing as well. The experiences with these tests have lead to a general concept of a state machine, which can be synthesized without problems. This concept guarantees a clock frequency as high as possible and is discussed in section 6.10.

### 4.3. Conclusions with regard to synthesis tools

There are synthesis and timing problems, which occur in both synthesis tools. An overview of all problems is given in the following list:

1. One specified rising edge notation.
2. Ranging of signals and variables.
3. INOUT variables.
4. Resource sharing.
5. Probing signals.
6. Implementation of type conversion functions.
7. Relational operators ($<$, $>$,$<=$,$>=$).
8. Construct of a state machine.
9. Replacement of records by variables; this replacement is necessary simply, because the synthesis tools do not support record types yet.
10. Vectors used to describe memory; as discussed before this solution requires replacement of the original VHDL code and a solution for this problem is discussed in the Master’s Thesis of K.J. Lammers, see [19].

Besides all these problems, the following conclusions are drawn:

- It is very important for a designer to know the hardware implementation of the used VHDL constructs. If the designer describes a design according to the guidelines of the synthesis tools and the recommendations mentioned in this chapter, during synthesis less problems are encountered.
- Although it is desirable to use HL descriptions as input for the emulation flow, synthesis results are improved, if the description is on an RTL level.
• Antologic is a very comprehensive synthesis tool. The tool contains several synthesis and optimization options. CORE is a very compact synthesis tool. The tool only contains some basic synthesis and optimization options.

• Compilation, synthesis and optimization take a great deal of time in Antologic. CORE synthesizes and optimizes fast.

• Comparison of the mapping results of both synthesis tools shows that synthesized designs of CORE are smaller than the synthesized designs of Antologic. In appendix F a table is listed with synthesis results of CORE and Antologic.

Next chapter discusses solutions for all mentioned problems.
5. Solution for synthesis and timing problems

After the disappointing experiences with the synthesis tools, it appears to be necessary to change the concept of the emulation flow. The conclusions in the previous chapter indicate that the level and style of VHDL is very important and that there is a gap between the HL description and the preferred descriptions for synthesis tools.

The missing part in the emulation flow is a construct that adapt the HL description to the preferred descriptions for synthesis tools. The synthesis problems and recommendations, mentioned in the previous chapter, lay the foundation for such a construct. This construct can be divided into the following parts:

1. Definition of templates.
   A template consists of a VHDL framework containing guidelines to describe a particular design block according to synthesis rules. Templates can be very helpful for the designer. They can guide the designer in writing his HL description. A template is written for several design blocks, for instance a template to describe a state machine description in VHDL.

2. Development of a Design Style Assistant tool (DSA tool). This tool has two main functions:
   a. Automatic verification of the VHDL code according to the templates.
      The large part of the function of this tool is based on the templates. A tool that checks the templates is not available on the professional software market, so the only solution is to develop such a DSA tool. The tool verifies if the design complies with the templates and if possible the tool changes the VHDL according to the templates. Some adaptations are made in interaction with the designer.
   b. Automatic replacement of VHDL code.
      Some of the problems mentioned in the previous chapter could be solved without interaction with the designer. In those cases the tool automatically replaces VHDL code. For instance the replacement of record types by variables is a problem that can be solved without interaction with the designer.
Taking this construct into account, the concept of the emulation flow has to be changed. In figure 9, the new concept of the emulation flow is shown. In this flow the DSA tool is inserted before the synthesis tool.

The DSA tool verifies the simulated HL description written by the designer according to the templates. If constructs like records are used in the description, the tool replaces the VHDL code. The working of the tool is symbolized by a funnel, see figure 10. This figure shows an HL description (written in the tool dependent subset and the PCALE subset) which is projected on the new HL description according to the DSA subset. This projection is controlled by the DSA tool.
If the tool generates errors or warnings, the designer has to review his HL description and start over with the complete design trajectory. If the verification is correct the VHDL has to be simulated again to verify if the adaptations made by the tool are correct. These simulation outputs can be compared with the simulation results obtained by simulating the HL description. Only if these results are identical the VHDL can be used as input for the synthesis tool. The HL description has been projected on the DSA subset and contains synthetizable VHDL. All known problems during synthesis have been solved by the DSA tool.

However, it is still possible that errors occur during synthesis, because of the following reasons:

- It is always possible to describe constructs (within the DSA subset) in the HL description which have such a complex hardware implementation that the synthesis tool crashes when synthesizing or optimizing the design.

- Synthesis tools are still under development, so it can occur that a bug in the tool causes a crash. During evaluation of the synthesis tools, several bugs have been found.

- Although the DSA tool has been tested thoroughly, it can occur that a bug in the DSA tool causes a crash.

- It is also possible that the system requirements of the synthesis tool are exceeded. If a design requires more memory space than available, the synthesis tool crashes.

Even after a successful synthesis and optimization run, the mapping could be a problem due to the fact that the logic, which has to be mapped, is too complex. The complexity of a design has a negative influence on the timing. So, it is possible that a complex design cannot be mapped at all even if a design is mapped successfully, it could be that the design does not function on a high frequency (at least 27 MHz).
In both cases, a possible solution is that the designer reviews the VHDL description and partitions the design in smaller parts separated by buffering. The designer has to start over with the total emulation flow. However, buffering causes a problem since it introduces a difference between the EPLD implementation (bread board) and the ASIC implementation of a design: in ASIC buffering is unnecessary. Therefore, further research is necessary to find other solutions to overcome these timing problems. The mapping tool and the timing problems in an EPLD are discussed in chapter 8.

The parts of the flow that are not discussed in this section, have already been described in section 2.3 and are unaltered.
6. Definition of templates

A formal specification of the DSA tool is written in some general templates. The decision to use templates to specify the DSA tool was taken because of the following advantages:

1. By forcing the designer to use the templates, one of the tasks of the DSA tool is verifying if the design complies with the templates. This simplifies the development of the DSA tool otherwise it would be necessary to implement an architectural synthesis tool. Another task of the tool is automatic replacement of particular code, for instance replacement of record types by variables. The total specification of the DSA tool is discussed in chapter 7.

2. A new subset of VHDL which can be used to describe designs and to enter synthesis tools, can be clearly defined. In chapter 3 the tool dependent subset was defined. This subset is refined again to a new, clearly defined subset: the DSA subset. This subset is described by the templates. In figure 11, the relation between the earlier defined subsets and the new subset is shown.

![Figure 11. The DSA VHDL subset](image)

In section 4.3 all synthesis problems found during testing of synthesis tools, are listed. According to this list, the following templates are defined in sections. These templates describe parts of the dataflow path of a design.
6.1. One specified rising edge notation

Each synthesis tool prefers another rising edge specification. The following template defines a standard rising edge declaration. Initializations of signals and variables are ignored by the synthesis tools. The template shows where the initialization of signals and variables has to be stated.

In relation with a rising edge a synchronous or an asynchronous reset can be defined. The template in figure 12, shows the best way to do this (according to the guidelines of the synthesis tool).

Template Rising Edge and Reset:
-- Initialisation in a declaration is not permitted except for constants.
-- Initialisation of signals, variables etc. can be achieved by adding
-- a reset signal, in one of the two following ways:

-- Synchronous reset
PROCESS
    ... -- do not initialise signals and/or variables here
BEGIN
    WAIT UNTIL (clk'EVENT AND clk = '1');
    IF reset = '1' THEN
        ... -- initialisation of signals, variables
    ELSE
        ...
    END IF;
END PROCESS;

-- Asynchronous reset
PROCESS
    ... -- do not initialise signals and/or variables here
BEGIN
    WAIT ON clk,reset;
    IF reset = '1' THEN
        ... -- initialisation of signals, variables
    ELSIF (clk'EVENT AND clk = '1') THEN
        ...
    END IF;
END PROCESS;

FIGURE 12. Template rising edge and reset
6.2. Ranging of signals and variables

Signals and variables of the integer or vector type have to be ranged. If the range of an integer is not specified the tool uses the default value of 32 bits for comparators, adders, substractors, etcetera. Otherwise, if the range is specified, the tool uses:

\[ \lfloor \log_2 (y - (x + 1)) \rfloor \] bits for integers of range x to y.

\[ y - x + 1 \text{ bits for vectors of range x to y (or range y downto x).} \]

The way signals and variables must be ranged is shown in template figure 13.

```
Template signals/variables:
  -- Define y > x (otherwise it is useless), then
  -- a synthesis tool will use \[ \lfloor \log_2 (y - (x + 1)) \rfloor \] bits for the declared integer
  VARIABLE an_int : integer RANGE x TO y;

  -- In case of the following declarations,
  -- a synthesis tool will use y - x + 1 bits for the declared std_ulogic_vector
  VARIABLE a_vector : std_ulogic_vector( x TO y);
  -- or:
  VARIABLE a_vector : std_ulogic_vector( y DOWNTO x);
```

**FIGURE 13. Template ranging signals and variables**

Template figure 13 applies for both signals and variables so the word SIGNAL can be replaced by the word VARIABLE.

6.3. Workaround for INOUT variables

Synthesis of procedures with INOUT variables, used to define static memory, appears to be a problem for the evaluated synthesis tools. Procedures with INOUT signals, to describe bidirectional signals, are synthesized correctly. In the following example a buffer is described:

```
PROCEDURE example_inout ( in_buffer : IN std_logic_vector(0 TO 10);
                          buffer      : INOUT std_logic_vector(0 TO 10);
                          out_buffer  : OUT std_logic_vector(0 TO 10)
                     ) IS

BEGIN
  out_buffer:= buffer;  -- value in previous clock period
  buffer:= in_buffer;   -- clock new data into buffer
END example_inout;
```
According to the evaluated synthesis tools, subprograms can only be used to describe combinational logic. Although this applies to most cases, the example above shows that sometimes INOUT variables have to be implemented with memory and that is the reason why synthesis of this kind of INOUT variables causes problems. When a procedure contains an INOUT variable and it is not intended as a memory for the next clock period, separation is not necessary. When the INOUT variable is intended as a memory for the next clock period, the INOUT variable has to be separated into an IN and an OUT variable; after the procedure-call, the OUT variable must be fed back to the IN variable, see template in figure figure 14.

Another rule of the evaluated synthesis tool, Autologic about implementing memory, is the following:

If a variable is read after it is set in a clock cycle, it is never necessary to allocate memory. For signals a memory element would be required in case the wait statement contains a condition clause (rising edge definition). (rule A)

This rule is further referred to as rule A. How the synthesis tools interpret this rule is explained in the following example:

```
PROCESS
    VARIABLE a: integer RANGE 0 TO 15;
BEGIN
    -- rising edge
    WAIT UNTIL (clk = '1' AND clk'EVENT);
    a := 5;  -- variable a is set
    ... := a;  -- variable a is read
END PROCESS;
```

In this example, variable a is read after it is set in one clock cycle so the synthesis tool does not allocate memory for variable a. This rule must be taken into account when separating an INOUT variable into an IN variable and an OUT variable. The following VHDL code illustrates the total solution for the INOUT problem:

```
PROCEDURE example_inout ( in_buffer : IN std_ulogic_vector(0 TO 10);
    buffer_varIN : IN std_ulogic_vector(0 TO 10);
    buffer_varOUT : OUT std_ulogic_vector(0 TO 10);
    out_buffer : OUT std_ulogic_vector(0 TO 10)
) IS
BEGIN
    out_buffer := buffer_varIN;  -- value of previous clock period
    buffer_varOUT := in_buffer;  -- clock new data into buffer
END example_inout;
```
The following procedure call must be used:

```
-- procedure call
example_inout(inbuf, buffer_varIN, buffer_varOUT, outbuf);
buffer_varIN := buffer_varOUT;
...
```

This notation can be used to force the tool to allocate memory for the variable `buffer_varIN`. Further use of this variable should be avoided, because of the memory rule A, otherwise the template would not have the expected implementation, as shown in the code below:

```
example_inout(... , buffer_varIN, buffer_varOUT, ...);
buffer_varIN := buffer_varOUT;  -- buffer_varIN is set
...
```

Because `buffer_varIN` is read after it is set, no memory is allocated for this variable, according to the rule. Since the variable `buffer_varIN` is equal to the variable `buffer_varOUT`, `buffer_varOUT` should have been used in the example above and memory had been allocated for `buffer_var_IN`. The variable `buffer_varOUT` can be used as IN variable in other procedures, functions or statements.

Taking this observation into account the rule should be replaced by the following memory rule, to increase the performance of the applied synthesis tools (except CORE):

**If a variable is read before it is set in the code above,**
**it is necessary to allocate memory. (rule B)**

The following example shows the difference between the two rules:

```
PROCESS
  VARIABLE a: integer RANGE 0 TO 15;
BEGIN
  -- rising edge;
  WAIT UNTIL (clk='1' AND clk'EVENT);
  ...
  := a;  -- variable a is read
  a := 5;  -- variable a is set
  ...
  := a;  -- variable a is read
END PROCESS;
```
In this example, the synthesis tools should allocate memory for variable a. However, in the last statement, variable a is read after it is set. So according to rule A, no memory is needed. According to rule B, memory is allocated for variable a because this variable is read in the first statement and never set before in the code. Unfortunately most synthesis tools do not use rule B but rule A for allocating memory. Therefore, the solution for INOUT variables that imply memory, only works if the IN variable is never used after the OUT variable has been fed back to it. The resulting template is shown in figure 14.

<table>
<thead>
<tr>
<th>Template procedure:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PROCEDURE</strong> proc_name</td>
</tr>
<tr>
<td><strong>BEGIN</strong></td>
</tr>
<tr>
<td><strong>...</strong></td>
</tr>
<tr>
<td><strong>END proc_name;</strong></td>
</tr>
</tbody>
</table>

```vhdl
mem_var IN : IN a_type;
mem_varOUT : OUT a_type;
...
) IS

-- The procedure call then becomes:
...
proc_name(...,mem_varIN,mem_varOUT,...);
mem_varIN := mem_varOUT;
...

-- mem_varOUT can be used, mem_varIN not

-- In other procedures, functions or statements mem_varOUT can be used as IN variable.
```

**FIGURE 14. Template workaround for INOUT variables of procedures**

### 6.4. Resource sharing

In section 4.1.1, the necessity to force the designer to write VHDL code with resource sharing is discussed. Sometimes resource sharing could be obtained very easy in a CASE statement. If various WHEN clauses of a case statement contain the same operation and if the result of these operation is assigned to the same variable then resource sharing is possible. In case of resource sharing, operation statements should be stated below the case statement; instead of the operation variable, update statements should be used in the different clauses. If no operation is to be performed, this must be achieved by setting the operation variable (update_count) to an initial value (for instance zero in case of addition). This kind of resource sharing is explained in the following example:

-- Example: CASE 1 without resource sharing

```vhdl
VARIABLE sum, result : integer RANGE 0 TO 31;
```
CASE x IS
    WHEN 1 => result := sum + 2;
    WHEN 2 => result := sum + 4;
    WHEN 3 => result := sum;
    WHEN OTHERS => result := sum + 7;
END CASE;

The implementation of this VHDL code is shown in figure 15.

FIGURE 15. Implementation of example CASE 1 without resource sharing

The following VHDL code shows the same example, written according to the template of figure 17:

-- Example: CASE 1 with resource sharing

VARIABLE sum, update_sum, result: integer RANGE 0 TO 31;
...
CASE x IS
    WHEN 1 => update_sum := 2;
    WHEN 2 => update_sum := 4;
    WHEN 3 => update_sum := 0;
    WHEN OTHERS => update_sum := 7;
END CASE;
result := sum + update_sum;

The implementation of example CASE 1, written according to the resource sharing template, is shown in figure 16.

FIGURE 16. Implementation of example CASE 1 with resource sharing
Even in this simple example two addition operators can be saved. This template, see figure 17, also applies to state machine case statements.

Template case-statement 1:

```
VARIABLE count, update_count : integer RANGE x TO y;
CASE expression IS
  WHEN clause_1 => update_count := ...
  WHEN clause_2 => update_count := ...
  ... WHEN clause_n => update_count := ...
  WHEN OTHERS => update_count := ...
END CASE;

count := count + update_count; -- operator could be: *, /, -, +
```

**FIGURE 17. Template CASE-statement 1**

Besides the sharing of operations in the different clauses, assignments could also be shared. If different variables are assigned with the same expression, this expression should be assigned to a variable and this variable is used in the different clauses. The following example shows how to use this kind of resource sharing:

```
-- Example: CASE 2 without resource sharing
VARIABLE sum, a, b, c, d, tmp : integer RANGE 0 TO 31;

CASE x IS
  WHEN 1 => a := sum + 8;
  WHEN 2 => b := sum + 8;
  WHEN 3 => c := sum + 8;
  WHEN OTHERS => d := sum + 8;
END CASE;
```

The implementation of this VHDL code is shown in figure 15.

**FIGURE 18. Implementation of example CASE 2 without resource sharing**
The following VHDL code shows the same example, written according to the template of figure 20:

```
-- Example: CASE 2

VARIABLE sum,a,b,c,d,tmp: integer RANGE 0 TO 31;

tmp := sum + 8;
CASE x IS
  WHEN 1 => a := tmp;
  WHEN 2 => b := tmp;
  WHEN 3 => c := tmp;
  WHEN OTHERS => d := tmp;
END CASE;
```

The implementation of example CASE 2, written according to the resource sharing template, is shown in figure 16.

![Diagram of implementation](image)

**FIGURE 19. Implementation of example CASE 2 with resource sharing**

If the designer takes resource sharing into account, three addition operators can be saved.
Template case-statement 2:

VARIABLE count, update_count : integer RANGE x TO y;
...
tmp := expr_containing_an_operator; -- operator could be: *,-,+, ...
CASE expression IS
  WHEN clause_1 => ...
    ... := tmp
  WHEN clause_2 => ...
    ... := tmp
  ...
  WHEN clause_n => ...
    ... := tmp
  WHEN OTHERS => ...
    ... := tmp
END CASE;
...

FIGURE 20. Template CASE-statement 2

Parallel to the two CASE templates, the following two IF templates are written:

Template if-statement 1:

VARIABLE tmp, update_tmp : integer RANGE x TO y;
...
tmp := ...
IF condition_1 THEN ...
  update_tmp := ...
ELSIF condition_2 THEN ...
  update_tmp := ...
...
ELSIF condition_n THEN ...
  update_tmp := ...
ELSE ...
  update_tmp := ...
END IF;
tmp := expr_containing_update_tmp; -- operator could be: *,-,+

FIGURE 21. Template IF-statement 1
Template if-statement 2:

VARIABLE tmp, update_tmp : integer RANGE x TO y;
...
tmp := expr_containing_an_operator;  --operator could be: *, /, -, +
IF condition_1 THEN ...
  ... := tmp
ELSIF condition_2 THEN ...
  ... := tmp
...  
ELSIF condition_n THEN ...
  ... := tmp
ELSE ...
  ... := tmp
ENDIF;

FIGURE 22. Template IF-statement 2

It is possible to perform resource sharing in the conditions of an IF-statement. The same construct as discussed with the template above applies, see figure 23.

Template if-statement 3:

VARIABLE tmp : integer RANGE x TO y;
...
tmp := expr_containing_an_operator;  --operator could be: *, /, -, +
IF condition_1_containing_tmp THEN ...
ELSIF condition_2a_not_containing_tmp THEN ...
ELSIF condition_2b_containing_tmp THEN ...
...  
ELSIF condition_n_not_containing_tmp THEN ...
ELSIF condition_n_containing_tmp THEN ...
ELSE ...
ELSE ...
ENDIF;

FIGURE 23. Template IF-statement 3

One must take into account that both CASE statements and IF statements could be nested. A recursive algorithm has been developed to check resource sharing in nested CASE and IF statements, see documentation of DSA tool [20].
6.5. Probing signals

If a signal is assigned in a process to a signal of the port map, the synthesis tool generates a flip-flop to latch the process signal before the assignment and a flip-flop to latch the port map signal. This implies extra delay. If you want to probe a signal, and this signal is defined in the process, place the assignment outside the process body. The assignment becomes asynchronous and the synthesis tool allocates no memory.

```
ENTITY example IS
  PORT ( clk : IN std_ulogic;
         rst : IN std_ulogic;
         bitcount : OUT integer;
         ... );
END example;

ARCHITECTURE example_arch OF example IS
  ...
  SIGNAL sig_counted_bits : integer;
  ...
BEGIN
  PROCESS
  BEGIN
    WAIT UNTIL (clk'EVENT AND clk = '1');
    IF (rst = '1') THEN
      ...
    ELSE
      sig_counted_bits <= ...
      ...
    END IF;
  END PROCESS;
  bitcount <= sig_counted_bits; -- probe the signal sig_counted_bits
END example_arch;
```

FIGURE 24. Template probing signals.

6.6. Type conversion functions

Type conversion functions are needed to convert types, for instance std_logic_vectors to integer, because it is easier to describe arithmetic functions of a design with arithmetic types, such as integers, than using the hardware related types, std_logic_vectors. Type conversion functions could be written by the designer, but this results in complex hardware
implementations, see section 4.2.1. By using the predefined type conversions of the synthesis tool an enormous reduction of hardware can be obtained, since the synthesis tool implements these conversion functions with the predefined implementation.

The best way to include these functions is to copy the VHDL code of these functions into a package. This package has to be available in the simulation tool. The names of the type conversion functions may not be changed and are used in the VHDL code. Simulation of the design could be performed with this package. After simulation, the package can be removed because the synthesis tool recognizes the type conversion functions and implements them with the predefined implementation instead of synthesizing them. This way, complex hardware can be avoided for these conversion functions.

6.7. FOR loops

Avoid the use of FOR loops. FOR loops are helpful to describe a repeated logic structure in a design. However, the evaluation of the synthesis tools has shown that these synthesis tools do not always synthesize FOR loops efficiently or even correctly. Thus the use of a FOR loop can result in inefficient or even incorrect hardware. So the designer must be aware of the hardware implementation of a FOR loop and that is not desirable.

6.8. Relational operators

If a relational operator has to be used, try to use the /= or = operators instead of the <, <=, >, >= operators, since the first two operators take considerably less hardware than the latter four relational operators. For instance, a comparator which only has to check if a variable equals 200 is less complex than a comparator which has to check if a variable has a value below 200.

6.9. Record types

Synthesis tools are still under development so not all VHDL statements are supported. Record types are very helpful to describe a design but these types are not supported. A workaround for this is to replace a record type by variables. This could be done automatically by the DSA tool. The example below shows the replacement. VHDL code with record type:

```vhdl
TYPE statics_header IS RECORD
  payload_unit_start_indicator : std_ulogic;
  packet_register : std_ulogic_vector(12 DOWNTO 0);
  adaptation_field_control : std_ulogic_vector(1 DOWNTO 0);
END RECORD;

VARIABLE packet_regs : statics_header;
```
Definition of templates

packet_regs.packet_register(12 DOWNTO 8) := data_in(4 DOWNTO 0);

Replaced by:

-- Removed record type statics_header
...
-- Replacement of record type statics_header
VARIABLE packet_regs_payload_unit_start_indicator : std_ulogic;
VARIABLE packet_regs_packet_register : std_logic_vector(12 DOWNTO 0);
VARIABLE packet_regs_adaptation_field_control : std_logic_vector(1 DOWNTO 0);

packet_regs_packet_register(12 DOWNTO 8) := data_in(4 DOWNTO 0);

The examples are functionally identical. Of course it is also possible to use the record type in the heading of a procedure. In this case, the tool changes the procedure heading, the procedure body and all procedure calls.

6.10. Construct of a state machine

The control part of a design is often described by a state machine. Synthesis tools have many problems with the synthesis of a state machine. All these problems can be avoided by using one of the following two templates:

- Template for state machine, type Moore.

  In a Moore state machine, the outputs do not directly depend on the inputs, only on the present state. A CASE-statement is used to describe the behavior of the state machine.

- Template for state machine, type Mealy.

  In a Mealy machine, the outputs depend directly on both the present state and the inputs. A PROCESS is used to assign the next state signal to the present state signal. Another PROCESS is used to describe a CASE-statement to define the state and output assignments. This PROCESS is sensitive to the present state signal and all (asynchronous) input signals.

Both state machine templates contain several templates discussed before, for instance the rising edge and reset template, the resource sharing templates. The reset state must be used to initialize variables and signals. In the enumerated type all state names are defined. The synthesis tools assume that the first name is the name of the start-up state (like the simulator). In the CASE-statements all state transitions must be defined, otherwise the WHEN OTHERS clause must be used to ensure that the synthesis tool can perform the state-encoding.
The DSA tool checks the templates, but to be able to do so the tool has to know where a state machine description starts. The designer has to mark the start of the state machine with some comment. This feature of the tool is discussed in the next section in more detail. The VHDL template for the type Moore is shown in figure 25.

![Template state machine (Moore)](image)

In figure 26, the template for state machine type Mealy is shown.
Template state machine (Mealy)

ENTITY state_machine IS
PORT ( clk : IN std_ulogic;
       rst : IN std_ulogic;
       ... );
END state_machine;

ARCHITECTURE state_machine_arch OF state_machine IS
... TYPE state_type IS (reset, s0, s1, ... );
SIGNAL present_state, next_state : state_type;
...
BEGIN
PROCESS
BEGIN
WAIT UNTIL (clk'EVENT AND clk = '1');
IF (rst = '1') THEN
  present_state <= reset;
ELSE
  present_state <= next_state;
END IF;
END PROCESS;

PROCESS (present_state, inputs, ...)
BEGIN
  -- next state assignments
  CASE present_state IS
    WHEN s0 => ...
      next_state <= ...;
    WHEN s1 => ...
      next_state <= ...;
    ...
    WHEN reset => ...
      -- initialisation of signals, variables ...
      next_state <= ...
      -- state initialisation
    WHEN OTHERS => ...
      next_state <= ...
  END CASE;
  ...
  -- define resource shareable operations here
END PROCESS;
END state_machine_arch;

FIGURE 26. Template State Machine (Mealy)
7. The Design Style Assistant tool

The main purpose of the DSA tool is to check the input design file and to write a new synthesizable output file. The DSA tool can be controlled in several ways. First of all, the user can specify which files must be checked and in which order. The names of the input design files have to be listed in a file, called vhdl.files. If a design is described in multiple input files, the tool generates an output file for each input file.

It is also possible to control the DSA tool by means of a control file. The translation and verification of the input files can exist of several different actions. Some of these actions are done without interference of the user, other actions are done interactively with the user. These actions and some features, added to increase the flexibility of the tool, can be controlled in the control file. The next section discusses all possible options of the control file in detail.

Besides the synthesizable output file, the tool generates a transcript file that lists all actions of the tool. The transcript file also contains errors or warnings generated by the tool. The user can use this file to review which statements in the VHDL code caused the errors or warnings.

7.1. Tool control

The user can control the DSA tool by specifying parameters in the control file. All these parameters have default values, so they are optional and need not be passed down to the tool. Since the number of parameters can be rather large, the optional parameters are not passed to the tool by means of an argument list but by means of a control file. The name of the control file has to be declared in the first argument of the tool-call. The parameters are:

- **MOORE_STATE_MACHINE_DECLARATION_MARK**
  This parameter is used to specify where a state machine description, type moore, starts, because the tool needs to know where the moore state machine template has to be verified. The default value of this parameter is “moore_state_machine_description”.

- **MEALY_STATE_MACHINE_DECLARATION_MARK**
  This parameter is used to specify where a state machine description, type mealy, starts, because the tool needs to know where the mealy state machine template has to be verified. The default value of this parameter is “mealy_state_machine_description”.

- **TEMPORARY_FILES_EXTENSION**
  When performing some actions, the tool needs some temporary result files. The names of these files are equal to the names of the original files extended with an extension as indicated by this parameter. When this parameter is not set, a default of “.tem1” is assumed. The main purpose of this parameter is that the user has influence on the extension of the temporary file so that this file can be recognized in case of a crash.
• SYNTHESIS_FILES_EXTENSION
For each input file the tool produces a synthesis output file. These files have to be simu-
lated again to verify functional correctness. After simulation these files are used for
synthesis of the design. These files have names equal to the original file names with an
extension specified by this parameter.
The default value of this parameter is ".syn.vhdl".

• TRANSCRIPT_FILE
With this parameter the name of the transcript file is specified. All actions and possible
warnings and errors of the tool are listed in a transcript file. This way, it is possible to
view the behavior of the tool without rerunning the tool again. The default name of the
transcript file is: "dsa.script"

• OPTIONS
This parameter is used to indicate which actions have to be performed by the tool. The
possibilities are:
BIT_CONVERSION
RECORD_CONVERSION
RANGE_VARIABLES
CHECK_INITIALIZATION
CHECK_RESOURCE_SHARING
CHECK_STATE_MACHINE
CHECK_INOUT_VARIABLE
The meaning of these keywords are described in the next section. By adding “on” or
“off” behind the keyword the user can turn the subtools on or off. Default all actions are
performed, except bit conversion.

• BEGIN OPERATOR FUNCTIONS & END OPERATOR_FUNCTIONS
Between these two parameters the user defined functions can be added, e.g. an adder
function. The first parameter indicates the beginning of the list, the last parameter the
end of the list. The functions defined in the list are checked for resource sharing.

The following code shows an example of a control file:

# OPTIONAL PARAMETERS; NEED NOT BE SPECIFIED
#
# String to use to detect moore state machine; default = "moore_state_machine_description"
MOORE_STATE_MACHINE_DECLARATION_MARK fsm_control_moore
# String to use to check of state machine; default = "mealy_state_machine_description"
MEALY_STATE_MACHINE_DECLARATION_MARK fsm_control_mealy
# Extension for temporary files; default = ".tem1"
TEMPORARY_FILES_EXTENSION. tem1
7.2. Actions of the tool

The functions implemented in the DSA tool are:

1. Replacement of type std_(u)logic into type bit and type std_(u)logic_vector into type bit_vector.
   
   After testing of the synthesis tools Autologic and CORE, a few tests have been done with a test version of VHDLSyn, also a synthesis tool. This test version could only manipulate the types bit and bit_vector so a type conversion was needed for the type std_(u)logic and for the type std_(u)logic_vector. If BIT_CONVERSION is turned on, the tool replaces the types automatically.

2. Replacement of record type into variables.
   
   If the option RECORD_CONVERSION is turned on, the tool automatically replaces all record types defined by the designer by variables.

3. Check initialization of integers and vectors and reset declaration.
   
   This function is controlled by the option CHECK_INITLALISATION. The function checks if there are initializations of signals or variables in a procedure or function. If so, the tool removes the initialization and put an assignment of the initialization value to the variable just after the begin statement of the procedure or function.

   Next, the tool verifies the reset template and generates warnings or errors if the VHDL code does not comply with this template. Initializations in the process are moved to the reset state, as described in rising edge and reset template, see figure 12.
4. Interactive check of range of variables and signals.

If RANGE_VARIABLES is turned on, all variables and signals of type integer or vector have to be ranged. If a variable or signal is ranged, the tool calculates how many bits are used in the synthesis tool for the concerning variable or signal and ask the user if this range has to be maintained. If a variable or signal is not ranged, the tool asks which range the concerning variable or vector must get.

5. Check if resource shareable constructs are used.

According to the template for resource sharing, discussed in section 6.4, the tool checks the VHDL code for resource sharing if the option CHECKRESOURCE_SHARING is set to on. The tool generates a warning if resource sharing could be useful.

6. Check state machine declaration.

This function is controlled by the option CHECK_STATE_MACHINE. The VHDL code is verified by the tool according to the state machine template. The reset declaration is verified, resource sharing is verified. Besides the tool checks if every clause of the last CASE statement, which describes the state assignments, contains a state assignment.

7. Check if INOUT variables are used.

The option CHECK_INOUT_VARIABLE can be turned on, so that the tool generates a warning if an INOUT variable is used in a procedure. Automatically replacement of INOUT variables according to the procedure template, figure 14, is not implemented because this replacement requires adaptations to procedure headings. These adaptations affect the whole VHDL source and since this INOUT problem is just temporary, automatically replacement is not implemented.

Before one of these functions is performed the tool reforms the VHDL format into a handful internal format. Multiple variables declared on one line are copied and declared on multiple lines. The closing bracket of the entity is placed on the next line, this simplifies replacement algorithms. For instance, the code:

```vhdl
ENTITY demultiplexer IS
    PORT (Clock
        Data_valid, Sync
        Byte_Clk
        END demultiplexer;

    : IN std_ulogic;
    : IN std_ulogic_vector(7 DOWNTO 0);
    : IN std_ulogic);

is replaced by:

ENTITY demultiplexer IS
    PORT (Clock
        Data_valid
        Sync
        Byte_Clk
        END demultiplexer;

    : IN std_ulogic;
    : IN std_ulogic_vector(7 DOWNTO 0);
    : IN std_ulogic_vector(7 DOWNTO 0);
    : IN std_ulogic
    )
    END demultiplexer;
```

Development of an emulation flow
8. The Mapping tool

The synthesis tool synthesizes and optimizes a VHDL description to a network description. This network description is written in a conversion format: AHDL (Altera Hardware Description Language) or EDIF. The last part of the emulation flow concerns the mapping to an EPLD, see [17]. The tool used for mapping is called MAX+PLUS. MAX+PLUS can execute the following functions:

- Load a design into MAX+PLUS.
- Map a design to an EPLD.
- Analyse the timing of a design.
- Simulate a design to check functional correctness.
- Program an EPLD.

8.1. Load a design

The AHDL file generated by CORE or the EDIF file generated by Autologic is used as input for MAX+PLUS in the emulation flow. MAX+PLUS loads the AHDL or EDIF file and compiles the file to an internal format. If the synthesis tool has translated the code correctly then usually no errors occur during this compilation. However, if synthesized design is very complex, it can happen that the compiler generates the warning: “Logic is too complex”. In this case, the designer has to review the VHDL description and partition the design in smaller parts separated by buffering. The designer has to start over with the total emulation flow.

However, buffering causes a problem since it introduces a difference between the EPLD implementation (breadboard) and the ASIC implementation of a design; in ASIC buffering is unnecessary. The designer has to keep that in mind.

8.2. Map a design

MAX+PLUS tries to map the description into an EPLD. The mapping in MAX+PLUS can be done to MAX5000 series, MAX7000 series and FLEX8000 amongst others. During evaluation all mappings have been done to the MAX7000 series. The MAX7000 architecture includes the following elements:

- Logic Array Blocks: A logic array block contains several macrocells and interfaces to both the I/O control block and the Programmable Interconnect Array. An EPLD consists of several Logic Array Blocks.
- Macrocells: A macrocell consists of a logic array, a product-term select matrix and a programmable register.
- Expander product terms (shared and parallel): The shared and parallel expanders offer the possibility to use additional product terms to supply needed resources.
- Programmable Interconnect Array: This array is a programmable path that routes across the entire device and offers the possibility to connect any signal source to any destination on the device.

- I/O control blocks: These blocks allow each I/O pin to be individually configured for input, output or bidirectional operation.

In addition to these basic elements, the MAX7000 architecture includes four dedicated inputs that can be used as general purpose inputs or as high-speed, global control signals (Clock, Clear, and two Output Enable signals) for each macrocell and I/O pin, see figure 27.

8.2.1. Logic Array Block

In this figure, multiple LABs are linked together via the Programmable Interconnection Array (PIA), a global bus that is fed by all dedicated inputs, I/O pins and macrocells. All inputs to each LAB, except the global control signals, are fed by 36 signals from the PIA. There are several LABs on an EPLD, depending on the type.
8.2.2. Macrocell

A macrocell, shown in figure 28, can be configured for both sequential and combinational logic operation. The macrocell consists of three functional blocks:

- The logic array.
- The product-term select matrix.
- The programmable register.

Combination logic is implemented in the logic array, which contains five product terms. The product-term select matrix allocates these product terms for use as either primary logic inputs (to the OR and XOR gates) to implement combinational functions, or as secondary inputs to the macrocell's register Clear, Preset, Clock and Clock Enable control functions. One product term per macrocell can be inverted and directly fed back into the logic array. This shareable product term can be connected to any other product term within the LAB. Based on the logic requirements of the design, MAX+PLUS automatically optimizes product-term allocation.

8.2.3. Expander product terms

Although most logic functions can be implemented with the five product terms available in each macrocell, some logic functions are more complex and require additional product terms. Instead of using another macrocell to supply needed resources, MAX+PLUS offers both shared and parallel expander product terms ("expanders") that provide additional product terms directly to any macrocell in the same LAB. These expanders help ensure that logic is synthesized with the fewest possible logic resources to obtain the fastest pos-
sible speed. Shareable expanders can be viewed as product terms (one from each macrocell) with inverted outputs that feedback into the logic array, see figure 29.

Parallel expanders are unused product terms from macrocells in the LAB outputs. the product terms select matrix can allocate parallel expanders to any neighboring macrocell (maximum 3) to implement fast, complex logic functions, see figure 30. The use of parallel expanders can be forced by the user, and results in a faster implementation of the EPLD.
8.2.4. Programmable Interconnect Array

Logic is routed between LABs on the Programmable Interconnection Array (PIA). This global bus is a programmable path that routes any signal source to any destination on the device. All dedicated inputs, I/O pin feedbacks, and macrocell feedbacks feed the PIA and route across the entire device. Only the signals required by each LAB are actually routed from the PIA into the LAB. While the routing delays of channel-based routing schemes in masked or field-programmable gate arrays are cumulative, variable and path-dependent, the PIA of MAX+PLUS has a fixed delay. The PIA eliminates skew between signals, and makes timing performance easy to predict.

Although synthesis tools try to predict the timing of a design by taking in account the structure of the back-end, the synthesis tools cannot predict the timing accurately because they only translate the design to a conversion format, AHDIl or EDIF, and this format does not contain any information about the routing. When this conversion format has been mapped to an EPLD, the routing is known and MAX+PLUS can predict the delay and timing accurately.

8.2.5. I/O control blocks

The I/O block, shown in figure 31, allows each I/O pin to be individually configured for input, output, or bidirectional operation. All I/O pins have a tri-state buffer that is controlled by one of two global active-low Output Enable pins (OE1 and OE2) or directly connected to GND or VCC. When the tri-state buffer control is connected to GND, the output is tri-stated (high impedance) and the I/O pin can be used as a dedicated input. When the tri-state buffer control is connected to VCC, the output is enabled. MAX7000 architecture provides dual I/O feedback, in which macrocell and pin feedbacks are independent. When an I/O pin is configured as an input, the associated macrocell can be used for buried logic.
8.3. Analyse the timing of a design

In order to verify on which clock frequency a design functions correctly, the timing analyser of MAX+PLUS can be useful. As mentioned before, MAX7000 devices have fixed internal delays that allow the user to determine the worst-case timing of a design. This part of the tool makes a good estimation the maximum clock frequency of a mapped EPLD.

8.4. Simulate a design

Simulation of the mapped design is possible with the MAX+PLUS simulator. An ASCII vector file can be used to describe the input signals. The simulation results can be compared with the simulation results obtained by simulating the High Level description in Vantage (or any other VHDL simulator), in order to verify functional correctness.

8.5. Program an EPLD

MAX+PLUS can program and erase an EPLD. The tool tries to map a design in the smallest EPLD. If the design does not fit in that EPLD, the tool automatically selects a larger one. The EPLDs in which a design has to fit, can be constrained by the user. If none of the EPLDs meets the requirements, the design is mapped in more EPLDs by partitioning the design. After mapping & programming, the programmed EPLD can be emulated.
9. Testing

The correctness of the developed DSA tool has to be tested in two ways:

1. Correctness for simulation; The replacements made by the DSA tool could affect the functional performance, so the changed code has to be simulated again to verify if functional behavior of the design is still correct.

2. Correctness for synthesis; If the tool does not generate any warnings or errors, synthesis should cause less problems (As mentioned before in chapter 5, a perfect synthesis run cannot be guaranteed).

Of course, DSA has to be tested to check its function in daily life practice. The DSA tool is first to be used in the current project of the Digital Video Processing (DVP) group at PCALE: a Demultiplexer/Descrambler IC as part of Digital TV Receivers. Therefore the Demultiplexer/Descrambler is used as the final testcase for the development of the DSA tool. The next section discusses this testcase. The remaining section discusses the correctness of the DSA tool. For a complete description of the Demultiplexer/Descrambler, the interested reader is referred to [6].

9.1. Testcase

At the present day, there is a worldwide race towards digital TV transmission systems. This race was triggered by the development of digital image compression standards. Among these standards are two standards defined by the Moving Pictures Expert Group (MPEG) from the International Standards Organization (ISO): the MPEG-1 standard and the MPEG-2 standard.

The MPEG-1 standard is a digital image compression algorithm originally intended for digital storage media. MPEG-1 is capable of reproducing full motion video at bit rates around 1.5 Mbit/s. The MPEG-1 standard is aimed at non-interlaced systems.

MPEG-2 is an extension to MPEG-1 in the sense that it enables full motion image reproduction at bit rates up to and including 15 Mbit/s (hence resulting in a higher image resolution). The MPEG-2 standard is aimed at digital TV broadcast systems. Since most broadcasting systems are interlaced, MPEG-2 is better suited for broadcasting systems than MPEG-1.

Note that both the MPEG-1 and MPEG-2 standard do not define an implementation. Only the syntax and the semantics of digital image compression are defined by these standards. For an extensive description of the MPEG-1 and MPEG-2 standards, see [4] and [5].

Apart from a reduction in bandwidth requirements through image compression, a digital TV broadcast system involves multiplexing and modulation. For digital TV broadcasting, satellite, cable and terrestrial transmission are considered. For the various transmission media, different modulation forms are envisaged. Furthermore, as each of the media has its own specific error characteristics, various channel coding methods such as Reed-Solo-
mon or Viterbi, are considered. Multiplexing is the technique behind the combination of video, audio and text services into a single bit stream. In present day TV systems Frequency Division Multiplexing (FDM) techniques are being used for this purpose. However, digital TV broadcasting has a tendency towards Time Division Multiplexing (TDM). Video, audio and text data are carried in fixed length packets. These packets are broadcast in random succession. The transmission order of the packets largely depends on the amount of channel capacity each individual service requires.

The combined Demultiplexer/Descrambler that is currently being developed by the DVP group at PCALE, is intended for use in MPEG-2 based Digital TV Receivers, possibly incorporating conditional access. Such receivers are to be implemented for instance in a Digital Video Broadcasting (DVB) top set box or in an integrated Digital TV Receiver. To get an idea of such applications, an example of a Demultiplexer/Descrambler system configuration is shown in figure 32.

Apart from the Demultiplexer/Descrambler unit itself, this configuration contains a channel decoder module consisting of a demodulator and a forward error corrector, source decoders A and B, a system micro-controller (μC) and a conditional access system. The main function of the Demultiplexer/Descrambler is to separate relevant data from an incoming data stream and pass it on to both the individual source decoders and the system micro-controller. In addition, parts of selected data streams can be descrambled, either internally or externally. For this purpose, the Demultiplexer subsystem contains the descrambler part of a conditional access system.

For more detailed description of these blocks the reader is referred to appendix D. In this appendix, the Demultiplexer/Descrambler is looked into in more detail.
Four blocks of the demultiplexer have been used to test if the DSA tool is correct for simulation and for synthesis. The blocks used to test the DSA tool are:

- MPEG-2 syntax parser
- Error handler including a Program Specific Information filter
- External descrambler interface
- Micro-controller interface

9.2. Testing the DSA tool

In order to test the DSA tool, the four blocks of the demultiplexer, mentioned in the previous section, have been tested. An overview of how the DSA tool is tested is depicted in figure 33.

![Diagram](image-url)

**FIGURE 33. Test to prove correctness of the DSA tool**
First of all, all blocks were simulated in the Vantage VHDL simulator and the simulation results were stored. After this simulation, the code was verified by the DSA tool. The tool made all necessary replacements and generated eleven warnings and one error. The total transcript file of the DSA tool for this test is listed in appendix G. Most warnings were generated to show the user that resource sharing might be possible.

An example of such a warning is shown in figure 34. In this example the variable `sect_byte_cnt` is assigned the result of an addition in two THEN clauses of an IF statement. Hence resource sharing is possible.

```
IF (load_byte_cnt = '1') THEN
  sect_byte_cnt := To_Integer( drive(section_length) ) + 3;
ELSIF (strt_priv_dat = '1') AND (pes_flt_act = '1') THEN
  sect_byte_cnt := To_Integer( drive(private_length) ) + 1;
ELSIF (sect_byte_cnt /= -1) THEN
  sect_byte_cnt := sect_byte_cnt - 1;
END IF;
```

**WARNING:** Resource sharing could be useful in expression:

```
sect_byte_cnt := To_Integer( drive(section_length) ) + 3 and
sect_byte_cnt := To_Integer( drive(private_length) ) + 1
```

**FIGURE 34.** Example of warning on resource sharing

This is just a warning and it is not necessary to adapt the VHDL code. However, if the designer adapts the code according to the templates for resource sharing, (see figure 17 to figure 23), a more optimal implementation can be established.

Other warnings occurred, checking the state machine description, see figure 35. This figure shows that the state variable `ts_af_det_state`, used to describe the state transitions, is not completely specified in the WHEN clause `af_length`. The state machine templates of figure 25 and figure 26 prescribe that the state transitions have to be specified completely. The designer can easily comply with these templates by adding an assignment to `ts_af_det_state` in this WHEN clause.

February 3, 1994
CHECKING STATE ASSIGNMENTS OF STATE ts_af_det_state

WHEN af_length =>
length fld ind := '0';
IF (Pwr Up Rst /= '0') THEN
  ts_af_det_state := af_length;
ELSIF (strt_af_parser = '1') THEN
  ts_af_det_state := flags;
ELSE
  NULL;
END IF;
WHEN flags =>

WARNING: State ts_af_det_state is not set in WHEN-statement

FIGURE 35. Example of warning on check state machine

One of the blocks of the demultiplexer causes an error. This block, the Program Specific Information filter, contains a variable that was initialized in a PROCESS. According to the rising edge and reset template of figure 12, initialization of variables and signals is not allowed in a PROCESS. The error message of the DSA tool is shown in figure 36.

However this block could not be tested because it contains a large register. How designs with large registers have to be emulated is described in the Master’s Thesis of K.J. Lammers, see [19]. The remaining part of the test was done without the Program Specific Information filter.
February 3, 1994

Txt

BEGIN

Is a reset declaration defined in entity (y/n)? n

+ ERROR: Initialisation of variables in a process is only permitted if a reset is used for this purpose
+ Synthesis tools cannot guarantee initial values other than through a reset signal,
+ so a reset signal must be added

FIGURE 36. Example of initialization error

The DSA tool generated a new, synthesizable file that contains all adaptations. This output file has also been simulated in Vantage. The simulation results have been compared with the stored simulation results of the original design and they were identical. This way correctness of the DSA tool for simulation was proved.

Next the output files were synthesized and optimized by the synthesis tool without any problem. This means that the templates and the DSA tool covered all known synthesis problems. The results of the synthesis step were mapped into an EPLD. This mapping also caused no problem. This whole test was performed in less than a day.

Of course, if the DSA tool generates more warnings and errors because there are more violations of the templates, the throughput time of the emulation flow increases considerably. However, if the design complies with the templates from the start, violations of the templates do not occur and the emulation flow can be traversed very fast. If the designer violates the templates, the DSA tool generates warnings and errors. The VHDL code must be adapted and the total time to map a design will take considerably more time.
10. Conclusions and recommendations

10.1. Conclusions

An emulation flow has been developed. The foundation of this flow is a VHDL synthesis tool. The performance of synthesis tools has been evaluated by synthesizing designs written according to the PCALE VHDL subset. This evaluation leads to four important conclusions:

- VHDL used as High Level Synthesis language offers the designer too much freedom in description style. Even VHDL written according to the PCALE subset can cause problems during synthesis.
- High Level Synthesis involves various VHDL subsets, for instance the PCALE subset and the tool supported subset. Many synthesis problems stem from the fact that these VHDL subsets have non-overlapping parts.
- VHDL synthesis tools show increased performance if a design description is written on RTL level.
- It is very important to know the hardware implementation of the used VHDL constructs, to avoid complex implementations.

Problems related to these conclusions have been solved by defining templates and developing a Design Style Assistant tool. The templates show which VHDL constructs a designer has to use in order to write synthesizable VHDL. The DSA tool verifies if the design complies with the defined templates and automatically replaces some unsupported constructs. By making this tool part of the emulation flow the known synthesis problems can be located and solved so that the performance of the emulation flow increases. Some other conclusions related to the flow can be drawn:

- The developed emulation flow is a useful extension to the Existing PCALE Design Flow, since emulation offers the possibilities of fast-prototyping, fast and cheap production start, field-testing and real time simulation.
- The DSA tool has been verified and is correct.
- If the design complies with the templates, the development of an emulation system takes considerably less time. This means that the defined templates solve the detected problems. Furthermore, the DSA tool checks the templates correctly and the use of all templates together enables a good mapping in EPLDs.
- Even after a correct mapping in an EPLD, it is possible that timing constraints cannot be met. This situation occurs when:
  1. the internal mapping in an EPLD causes too much delay.
  2. the synthesis tool implements a design as a very complex structure. Complex structures always cause much delay after mapping into an EPLD.
  3. the designer has written VHDL according to the defined templates, but has still used structures that require too complex logic.
10.2. Recommendations

The developed emulation flow can be improved. Possible refinements of the emulation flow are summarized below:

- The emulation flow can only be successful if the synthesis tool synthesizes correctly. Synthesis tools can be improved on two points:
  1. The supported VHDL subset can be extended.
  2. Implementation algorithms used by synthesis tools are not optimal and sometimes not even correct. So, better implementation algorithms would improve the performance of a synthesis tool.

- One of the evaluated synthesis tools was a specific synthesis tool. A specific synthesis tool is a synthesis tool that has synthesis algorithms specific for each target implementation. The other evaluated synthesis tool was a generic synthesis tool. A generic synthesis tool does not have specific synthesis algorithms for each target implementation, it has general synthesis algorithms. Comparing the specific synthesis tool with the generic synthesis tool, it is noticed that the specific synthesis tool produces better synthesis results. If this statement holds for generic and specific synthesis tools in general, is to be investigated.

- A new version of a synthesis tool differs in performance in relation to an old version. So, a new version has to be evaluated again. It is possible that the defined templates or the DSA tool have to be changed. Therefore, the templates and the DSA tool have to be kept up-to-date with the state of the synthesis tool.

- The developed emulation flow does not fully comply with the philosophy of the PCALE design flow. The PCALE philosophy requires functional verification on every level. However, the network description cannot be checked for functional correctness because this description cannot be simulated. The possibility to simulate the synthesis results is desirable, since it would complete the developed emulation flow according to the PCALE design flow philosophy.

- In case synthesis is performed with EPLDs as back-end in mind, the synthesis tool and the mapping tool could be more integrated. The synthesis tool translates a design into a conversion format, this conversion format is compiled and translated again to an internal format by the mapping tool. If the synthesis tool and the mapping tool are integrated, translations and compilation become redundant. Integration of the tools does not mean that the consecutive steps are integrated, merely, synthesis with ASIC in mind still remains possible.

- The presentation of the templates could be improved for instance by use of a Schematic Entry tool. A Schematic Entry tool is based on a graphical environment. The templates could be symbolized by a button. If the user selects this button the text of the template is shown.
Appendix A. List of References

[1] Hurk, J.A.A.M van den
   Evaluation of the PCALE VLSI Design Flow for HDTV ICs.
   Report of a second phase project.

   New York, The Institute of Electrical and Electronics Engineers, Inc.,

[3] Lakerveld, J.
   VHDL modelling guidelines for Digital TV Receiver project v1.0.
   Internal Report, number: ETV/IR93038.
   Eindhoven, Product Concept and Application Laboratory Eindhoven,
   April 1993.

   ISO/IEC DIS 11172

   ISO/IEC JTC 1/SC 29 N 658
   December, 1993.

   Tentative Device Specification of the MPEG-2 Demultiplexer/Descrambler.
   Internal Laboratory Report.
   Eindhoven: Philips Semiconductors, Dept. PCALE,
   November 8th, 1993. Draft version 2.0

[7] Horvath, T.A.
   Hardware emulation of VLSI design
   Proc. Third Annual IEEE ASIC Seminar and Exhibit,

[8] Tuck, B.
   Are designers' expectations for synthesis realistic?
   Computer Design (USA), Vol. 29(1990), Iss. 21, p. 110-112, 114, 118, 120, 126, 128, 129.

   Wilsonville, Oregon: Mentor Graphics Corporation,
   1993. Software Version 8.2

    Wilsonville, Oregon: Mentor Graphics Corporation,
    1993. Software Version 8.2

    Suite 102, Berkeley, CA: Exemplar Logic,
    1993. Software Version 1.21
   Suite 102, Berkeley, CA: Exemplar Logic,
   1993. Software Version 1.21

    VHDL Syn v1.0 User Manual.
    Hilversum, (The Netherlands). Philips Electronic Design & Tools,

    HDL-Oriented Synthese-Methodik. Teil II.

    VHDL as input for High-Level Synthesis.

[16] Nagasamy, V. et al.
    Specification, Planning and Synthesis in a VHDL Design Environment.

    San Jose, CA: Altera Corporation,
    1992. Software Version 3.0

    Femont, CA: Vantage Analysis Systems,

[19] Lammers, K.J.
    Development of an emulation flow for designs with large memory requirements.
    Department of Electrical Engineering,
    Eindhoven University Of Technology, Design Automation Section,
    February 1994.

[20] Lieshout, L.P.M.
    Documentation of the DSA tool.
    Eindhoven, Product Concept and Application Laboratory Eindhoven,
Appendix B. Brief description of QDMC IC

The QDMC is the controller IC for the QDM (Quadrature Demodulator) and can be switched between QPSK and BPSK demodulation with a mode control pin. It serves:

1. Carrier Recovery Control. The Carrier Recovery synchronizes the frequency of the receiver oscillator to the transmitter carrier frequency.

2. Automatic Frequency Control (AFC). In case AFC is switched on, a track region is activated. This enables the system to track frequency deviation automatically.

3. Carrier Lock Indicator. The lock indicator indicates whether the loop is in lock or not.

4. Automatic Gain Control (AGC). This loop controls the amplitude of the incoming signal to a desired fixed value.

5. Clock Recovery for a QPSK/BPSK demodulator receiver system. An internal clock circuit is synchronized to the symbol clock of the input signal.

FIGURE 37. Application of QDMC IC
Appendix C. Supported VHDL subset of synthesis tools

The meaning of the used symbols is:
- **X**: in the no support column indicates an item that stops the compiler and generates an error message. In the support column, X indicates that this VHDL construct is supported.
- **i**: in the no support column indicates an item that is ignored for synthesis.
- **R**: in the no support column indicates an item that is restricted; this item belongs to the subset: unsynthesizable VHDL.
- **U**: unknown; probably supported, reference does not clearly state the rules for synthesizability of this item.
- **n**: n is short for note: below table, a note about this VHDL construction is stated.

<table>
<thead>
<tr>
<th>Supported by</th>
<th>Autologic</th>
<th>CORE</th>
<th>VhdlSyn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Construct</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AFTER clause in concurrent signal assignment statement</td>
<td>X i</td>
<td>Ri</td>
<td>X</td>
</tr>
<tr>
<td>AFTER clause in sequential signal assignment statement</td>
<td>X i</td>
<td>Ri</td>
<td>X</td>
</tr>
<tr>
<td>ALIAS declaration</td>
<td>X</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>Architecture body</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Assertion statement (concurrent)</td>
<td>X i</td>
<td>R i</td>
<td>X</td>
</tr>
<tr>
<td>Assertion statement (sequential)</td>
<td>X i</td>
<td>R i</td>
<td>X</td>
</tr>
<tr>
<td>Attribute declaration</td>
<td>X</td>
<td>X</td>
<td>U</td>
</tr>
<tr>
<td>Attribute specification</td>
<td>X</td>
<td>X</td>
<td>U</td>
</tr>
<tr>
<td>Block statement</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Bus signal</td>
<td>X</td>
<td>R i</td>
<td></td>
</tr>
<tr>
<td>Case statement</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Component declaration</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Notes table 2, CORE remarks: • Blocks cannot have ports or generic.
Table 3: Supported Constructs 2

<table>
<thead>
<tr>
<th>Supported by</th>
<th>Autologic</th>
<th>Core</th>
<th>VhdlSyn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Construct</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Null statement</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Package body / declaration</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Procedure / Procedure call</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Process statement</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Register signal</td>
<td>X</td>
<td>R i</td>
<td></td>
</tr>
<tr>
<td>Return statement</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Signal assignment, conditional (concurrent)</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Signal assignment, selected (concurrent)</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Signal assignment, simple (concurrent)</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Signal assignment (sequential)</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Signal declaration</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Subprogram body / declaration</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Subtype declaration</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Type conversion</td>
<td>X</td>
<td>X</td>
<td>U</td>
</tr>
<tr>
<td>Type declaration</td>
<td>X</td>
<td>Xn</td>
<td>U</td>
</tr>
<tr>
<td>Use clause</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Variable assignment statement</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Variable declaration</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Wait statement</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Notes table 3, CORE remarks:

- RETURN statement must be last in a function.
- Global, non-constant signals not supported.
- WAIT statements: multiple assignments not synthesizable.
- Type conversion only with textual enum.types
Table 4: Supported Constructs (continued)

<table>
<thead>
<tr>
<th>Supported by</th>
<th>Autologic</th>
<th>Core</th>
<th>VhdlSyn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Construct</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Component instantiation statement</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Configuration declaration</td>
<td>X</td>
<td>R</td>
<td>i</td>
</tr>
<tr>
<td>Configuration specification</td>
<td>X</td>
<td>X</td>
<td>i</td>
</tr>
<tr>
<td>Constant declaration</td>
<td>X</td>
<td>X</td>
<td>i</td>
</tr>
<tr>
<td>Disconnection specification</td>
<td>X</td>
<td>R</td>
<td>i</td>
</tr>
<tr>
<td>Entity declaration</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Exit statement</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>File declaration</td>
<td>X</td>
<td>R</td>
<td>X</td>
</tr>
<tr>
<td>Function / Function call</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Generate statement</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Generic clause</td>
<td>X</td>
<td>R</td>
<td>i</td>
</tr>
<tr>
<td>If statement</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Library clause</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Loop statement (without an iteration scheme)</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Loop statement with FOR clause</td>
<td>X</td>
<td>Xn</td>
<td>X</td>
</tr>
<tr>
<td>Loop statement with WHILE clause</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Next statement</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Notes table 4, CORE remarks:
- Configurations are ignored; default binding assumed.
- Loop variables have to be integers.

Table 5: Supported Operators

<table>
<thead>
<tr>
<th>Supported by</th>
<th>Autologic</th>
<th>Core</th>
<th>VhdlSyn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator class</td>
<td>Operators</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Miscellaneous</td>
<td></td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>**</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>abs</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Multiplying</td>
<td>/, mod, rem</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>* , sll, srl, sra, rll, rrl</td>
<td>X</td>
<td>Xn</td>
</tr>
<tr>
<td>Sign</td>
<td>+, -</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Adding</td>
<td>+, -, &amp;</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Relational</td>
<td>=, /=, &lt;, &lt;=, &gt;, &gt;=</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Logical</td>
<td>and, or, nand, nor, not, xor</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Development of an emulation flow 73
Notes table 5, VhdlSyn and CORE remarks:

- multiplying operator *.,/.mod.rem are supported for integers of constant power of 2.

### Table 6: Supported Attributes

<table>
<thead>
<tr>
<th>Supported by</th>
<th>Autologic</th>
<th>Core</th>
<th>VhdlSyn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object kind</td>
<td>Attribute Name</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Array</td>
<td>'high</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'left</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'length</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'low</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'range</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'reverse_range</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'right</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Block</td>
<td>'behaviour</td>
<td>X</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>'structure</td>
<td>X</td>
<td>R</td>
</tr>
<tr>
<td>Signal</td>
<td>'active</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>'delayed</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>'event</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>'last_active</td>
<td>X</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>'last_event</td>
<td>X</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>'last_value</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>'quiet</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>'stable</td>
<td>X</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>'transaction</td>
<td>X</td>
<td>R</td>
</tr>
<tr>
<td>Type</td>
<td>'base</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>'high</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>'left</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>'leftof</td>
<td>X</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>'low</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>'pos</td>
<td>X</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>'pred</td>
<td>X</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>'right</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>'rightof</td>
<td>X</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>'succ</td>
<td>X</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>'val</td>
<td>X</td>
<td>R</td>
</tr>
</tbody>
</table>
Table 7: Supported Types

<table>
<thead>
<tr>
<th>Supported by</th>
<th>Autologic</th>
<th>Core</th>
<th>VhdlSyn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Name</td>
<td>Y N</td>
<td>Y N</td>
<td>Y N</td>
</tr>
<tr>
<td>Access</td>
<td>X R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array</td>
<td>X Xn</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Bit</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Bit_vector</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Boolean</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Character</td>
<td>X R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enumerated</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>File</td>
<td>X X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating Point</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td>X X X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Natural</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Physical</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Positive</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Qsim_state</td>
<td>X Xn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Qsim_state_vector</td>
<td>X</td>
<td>Xn</td>
<td></td>
</tr>
<tr>
<td>Real</td>
<td>X R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Record</td>
<td>X R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Severity_level</td>
<td>X Xi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Std_ulogic</td>
<td>X X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Std_ulogic_vector</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>String</td>
<td>X Xi</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes table 7, CORE remarks:
- Multidimensional arrays supported up to two dimensions.
- Arrays of arrays not supported.
- Array indices have to be integers.
- String and character manipulations are not supported.
- Qsim_state and Qsim_state_vector not supported but package for conversion available.
Appendix D. The Demultiplexer/Descrambler

The internal structure of the MPEG-2 Demultiplexer/Descrambler is shown in the functional block diagram in figure 38. The block diagram indicates the main functional entities in the Demultiplexer/Descrambler.

The functional entities and their meaning are:

- **MPEG-2 syntax parser**
  The MPEG-2 syntax parser parses so-called transport streams that comply with the MPEG-2 Systems specification.

- **Error handling**
  Error handling is invoked whenever an error is detected.

- **Internal descrambler**
  The internal descrambler descrambles the incoming data stream.

- **External descrambler interface**
  The external descrambler interface is for the communication with an optional external descrambler device. The throughput delay of the external descrambler is compensated for in the interface module.
• Teletext filter
  The teletext (TXT) filter generates a teletext clock and provides a serial TXT data stream.

• High Speed data filter
  The High Speed (HS) data filter retrieves entire transport packets from the input stream; the filtered data is stored in a First In First Out (FIFO) buffer.

• Auxiliary data filter
  The auxiliary data filter derives data from the transport stream. Auxiliary data is protected by a Cyclic Redundancy Check (CRC) code, which is checked and removed by the filter.

• Application Data Filter 2
  This data filter in fact does not filter at all, it merely passes the entire transport stream on in byte format. In addition, a byte strobe signal (indicating consecutive bytes) and a header byte indicator signal are generated.

• Application Data Filter 1
  This data filter is intended for video data and has a vendor specific interface. It selects Packetized Elementary Stream (PES) data and passes it to the video FIFO buffer. Time-stamps are obtained from the PES stream, also.

• Application Data Filter 0
  As Application Data Filter 1, except that this filter is for audio data.

• Program Clock Reference processor
  The Program Clock Reference (PCR) processor is capable of regenerating a local system time clock. A local clock counter generates an absolute timing value which is used to verify the phase relationship between the local system time clock and the transmitter reference clock.

• Two time-stamp processors
  The time-stamp processors are for synchronization of the attached source decoders. These processors compare incoming time-stamps with the local absolute time value generated by the PCR processor. In case of equality an interrupt is generated and sent to the micro-controller (μC) for further handling.

• Two FIFO buffers with buffer control
  These buffers are intended for the interfacing between different clocking systems.

• Micro-controller interface
  The micro-controller (μC) interface provides protocol handling for the I/O bus and contains filters for retrieving program specific information and entitlement message data from the transport stream.
Appendix E. VHDL code of testcase State machine

Package:
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

PACKAGE transpack IS
  CONSTANT fixed_pckt_len : integer := 255;
  CONSTANT sync_len : integer := 8;
  CONSTANT scram_ctrl_len : integer := 1;
  CONSTANT continuity_len : integer := 3;
  CONSTANT flag_len : integer := 1;
  CONSTANT pckt_err_ind_len : integer := 1;
  CONSTANT pckt_id_prior_len : integer := 15;
  CONSTANT sync_byte : std_ulogic_vector(0 TO sync_len - 1) := "10001111";
  SUBTYPE integer_max IS integer RANGE 0 TO fixed_pckt_len;
  SUBTYPE integer_min IS integer RANGE 0 TO 40;
  TYPE state_machine IS (sync, scrambling_control, continuity_counter, flag_adaptation,
ind_pckt_err, pckt_id_prior, pckt_data, reset);
END transpack;

PACKAGE BODY transpack IS
END transpack;

Main file:
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY demux;
USE demux.transpack.ALL;

ENTITY t IS
  PORT ( clk : IN std_logic;
         rst : IN std_logic;
         data_in : IN std_logic;
         data_out : OUT std_logic;

Development of an emulation flow 79
ARCHITECTURE transstream_arch OF t IS
BEGIN
-- statemachine
proc1 : PROCESS
VARIABLE counted_bits : integer_min;
VARIABLE total_bits : integer_max;
VARIABLE update_total : integer_max;
VARIABLE sync_reg : std_ulogic_vector(0 TO sync_len - 1);
VARIABLE scram_contr_reg : std_ulogic_vector(0 TO scram_ctrl_len);
VARIABLE continuity_reg : std_ulogic_vector(0 TO continuity_len);
VARIABLE adaptation_fl : std_ulogic;
VARIABLE pkt_err_reg : std_ulogic_vector(0 TO pckt_err_reg - 1);
VARIABLE pckt_id_prior_reg : std_ulogic_vector(0 TO pckt_id_prior_len);
VARIABLE dump_reg : std_ulogic;
VARIABLE nextstate : state_machine;
VARIABLE cnt_rst : BOOLEAN;
VARIABLE cnt_en : BOOLEAN;
VARIABLE tot_cnt_rst : BOOLEAN;
VARIABLE tot_cnt_en : BOOLEAN;
BEGIN
    -- replace of rising edge;
    WAIT UNTIL (clk = '1' AND clk'EVENT);
    IF (rst = '1') THEN
        counted_bits := 0;
        total_bits := 0;
        nextstate := reset;
    ELSE
        CASE nextstate IS
            WHEN sync => sync_reg(sync_reg'LEFT TO sync_reg'RIGHT - 1) :=
                sync_reg(sync_reg'LEFT + 1 TO sync_reg'RIGHT);
                sync_reg(sync_len - 1) := data_in;
            WHEN scrambling_control =>
                scram_contr_reg(scram_contr_reg'LEFT TO
                    scram_contr_reg'RIGHT - 1) :=
                scram_contr_reg(scram_contr_reg'LEFT + 1 TO

Development of an emulation flow
scram_contr_reg(RIGHT);
scram_contr_reg(scram_ctrl_len) := data_in;

WHEN continuity_counter =>
  continuity_reg(continuity_reg(LEFT TO continuity_reg(RIGHT - 1)) :=
  continuity_reg(continuity_reg(LEFT + 1 TO continuity_reg(RIGHT));
  continuity_reg(continuity_len) := data_in;

WHEN flag_adaptation => adaptation_fl := data_in;
WHEN ind_pkt_err =>
  pkt_err_ind(pkt_err_ind_len - 1) := data_in;
WHEN pkt_id_prior =>
  pkt_id_prior_reg(pkt_id_prior_reg(LEFT TO
  pkt_id_prior_reg(RIGHT - 1)) :=
  pkt_id_prior_reg(pkt_id_prior_reg(LEFT + 1 TO
  pkt_id_prior_reg(RIGHT));
  pkt_id_prior_reg(pkt_id_prior_len) := data_in;

WHEN pkt_data =>
  IF total_bits <= fixed_pkt_len THEN
    data_out <= data_in;
  END IF;

WHEN OTHERS => dump_reg := data_in
END CASE;
update_total:=0;

CASE nextstate IS
WHEN sync =>
  IF (sync_reg = sync_byte) THEN
    nextstate := scrambling_control;
    cnt_rst:= TRUE;
    cnt_en:=FALSE;
    update_total:=sync_len;
    tot_cnt_en:=TRUE;
    tot_cnt_rst:=FALSE;
  ELSE
    nextstate := sync;
    cnt_rst:=FALSE;
    cnt_en:=FALSE;
    tot_cnt_en:=FALSE;
    tot_cnt_rst:=FALSE;
  END IF;
WHEN scrambling_control =>
  IF counted_bits = scram_ctrl_len THEN
    nextstate := continuity_counter;
    update_total:= 2;
    tot_cnt_en:=TRUE;
    cnt_rst:=TRUE;
    tot_cnt_rst:=FALSE;
    cnt_en:=FALSE;
  ELSE
    nextstate := scrambling_control;
    cnt_en:=TRUE;
    tot_cnt_rst:=FALSE;
    tot_cnt_en:=FALSE;
    cnt_rst:=FALSE;
  END IF;
WHEN continuity_counter =>
  IF counted_bits = continuity_len THEN
    nextstate := ftag_adaptation;
    update_total:= 4;
    tot_cnt_en:=TRUE;
    cnt_rst:=TRUE;
    cnt_en:=FALSE;
    tot_cnt_rst:=FALSE;
  ELSE
    nextstate := continuity_counter;
    cnt_en:=TRUE;
    tot_cnt_en:=FALSE;
    cnt_rst:=FALSE;
    tot_cnt_rst:=FALSE;
  END IF;
WHEN ftag_adaptation =>
  nextstate := ind_pkt_err;
  update_total:= flag_len;
  tot_cnt_en:=TRUE;
  cnt_rst:=FALSE;
  cnt_en:=FALSE;
  tot_cnt_rst:=FALSE;
WHEN ind_pkt_err =>
nextstate := pckt_id_prior;
update_total:=flag_len;
tot_cnt_en:=TRUE;
cnt_rst:=FALSE;
cnt_en:=FALSE;
tot_cnt_rst:=FALSE;

WHEN pckt_id_prior =>

IF counted_bits = pckt_id_prior_len THEN
   nextstate := pckt_data;
   update_total:=16;
   tot_cnt_en:=TRUE;
   cnt_rst:=TRUE;
   cnt_en:=FALSE;
   tot_cnt_rst:=FALSE;
ELSE
   nextstate := pckt_id_prior;
   cnt_en:=TRUE;
   cnt_rst:=FALSE;
   tot_cnt_en:=FALSE;
   tot_cnt_rst:=FALSE;
END IF;

WHEN pckt_data =>

IF total_bits = fixed_pkt_len THEN
   nextstate := sync;
   FOR i IN 0 TO sync_len - 1 LOOP
      sync_reg(i) := '0';
   END LOOP;
   tot_cnt_rst:=TRUE;
   cnt_rst:=TRUE;
   cnt_en:=FALSE;
   tot_cnt_en:=FALSE;
ELSE
   update_total:=1;
   tot_cnt_en:=TRUE;
   tot_cnt_rst:=FALSE;
   cnt_rst:=FALSE;
   cnt_en:=FALSE;
   nextstate := pckt_data;
END IF;
WHEN reset =>
    sync_reg := "0000000";
    nextstate := sync;
    tot_cnt_rst:=TRUE;
    cnt_rst:=TRUE;
    cnt_en:=FALSE;
    tot_cnt_en:=FALSE;

WHEN OTHERS => nextstate := reset;
END CASE;

IF cnt_rst = TRUE
    THEN counted_bits:=0;
ELSE
    IF cnt_en = TRUE THEN counted_bits:=counted_bits+1;
    END IF;
END IF;

IF tot_cnt_rst = TRUE THEN total_bits:=0;
ELSE
    IF tot_cnt_en = TRUE THEN total_bits:=total_bits + update_total;
    END IF;
END IF;

END IF;

END PROCESS proc1;

END transstream_arch;
Appendix F. Test results

All designs have been mapped to the EPLD type EPM7160LC84-2.

Table 8: Results of the synthesis of the state machine

<table>
<thead>
<tr>
<th>Synthesis Tool</th>
<th>Options</th>
<th>Pass</th>
<th>Soft cells</th>
<th>Logic cells</th>
<th>Shar. exp</th>
<th>Clk frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORE</td>
<td>3</td>
<td>-</td>
<td>78</td>
<td>4</td>
<td>25 MHz</td>
<td></td>
</tr>
<tr>
<td>max_pt = 32</td>
<td>2</td>
<td>-</td>
<td>111</td>
<td>10</td>
<td>26 MHz</td>
<td></td>
</tr>
<tr>
<td>max_pt = 32</td>
<td>3</td>
<td>-</td>
<td>120</td>
<td>25</td>
<td>13.5 MHz</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Synthesis Tool</th>
<th>Optimization Area</th>
<th>Optimization Speed</th>
<th>Soft cells</th>
<th>Logic cells</th>
<th>Shar. exp</th>
<th>Clk frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autologic</td>
<td>low</td>
<td>low</td>
<td>no</td>
<td>87</td>
<td>6</td>
<td>37 MHz</td>
</tr>
<tr>
<td></td>
<td>low</td>
<td>high</td>
<td>no</td>
<td>87</td>
<td>9</td>
<td>37 MHz</td>
</tr>
<tr>
<td></td>
<td>high</td>
<td>low</td>
<td>no</td>
<td>*</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>high</td>
<td>high</td>
<td>no</td>
<td>*</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>low</td>
<td>low</td>
<td>yes</td>
<td>71</td>
<td>45</td>
<td>20 MHz</td>
</tr>
<tr>
<td></td>
<td>low</td>
<td>high</td>
<td>yes</td>
<td>64</td>
<td>36</td>
<td>20 MHz</td>
</tr>
<tr>
<td></td>
<td>high</td>
<td>low</td>
<td>yes</td>
<td>59</td>
<td>28</td>
<td>25 MHz</td>
</tr>
<tr>
<td></td>
<td>high</td>
<td>high</td>
<td>yes</td>
<td>65</td>
<td>26</td>
<td>25 MHz</td>
</tr>
</tbody>
</table>

• * means error in MAX+PLUS "logic too complex"

• SOFT cells: Option of Autologic to turn on insertion of SOFT buffers in MAX+PLUS. A logic synthesis option that directs the MAX+PLUS Compiler module to insert SOFT buffers at advantageous locations in the project. If this option is turned on, illegal combinational feedback is automatically fixed. If this option is turned off, all gates -- except those with manually entered SOFT or LCELL buffers -- are flattened to sum-of-products form, and must fit in a single logic cell.

• max_pt = 32: Option of CORE to control size of product terms.

Using this option means that the tool will break every function wider than 32 product terms into more than one level, instead of the default value of 24. This results in a better mapping in MAX+PLUS, because the maximum clock frequency on which the design will function correct, will increase.
Appendix G. Testresult of the DSA tool

CASE i-1 IS
  WHEN 112141517181011111221326 =>
    crc32_reg(i) := crc32_reg(i-1) XOR feedback;
  WHEN OTHERS =>
    crc32_reg(i) := crc32_reg(i-1);
END CASE;

crc32_reg(1) := dl8_byte_out(j) XOR feedback;

WARNING: Resourcesharing could be useful in expression:
crc32_reg(i) := crc32_reg(i-1) XOR feedback and crc32_reg(i) := crc32_reg(i-1)

IF (strt_pu_parser = '0') AND (prev_strt_pu_parser = '1') THEN
  sect_strt_cnt := To_Integer(drive(eh_byte_out)); -- pointer
  IF (drive(eh_byte_out) = 0) THEN
    strt_sect_parser <= '1';
  ELSE
    strt_sect_parser <= '0';
  ENDIF;
  sect_strt_cnt := sect_strt_cnt - 1;
  sect_pld_ind <= '0';
ELSIF (sect_strt_cnt = 0) THEN
  strt_sect_parser <= '1';
  sect_strt_cnt := sect_strt_cnt - 1;
ELSIF (sect_strt_cnt = -1) THEN
  strt_sect_parser <= '0';
ELSE
  sect_strt_cnt := sect_strt_cnt - 1;
ENDIF;

WARNING: Resource sharing could be useful in expression:
sect_strt_cnt := sect_strt_cnt - 1 and sect_strt_cnt := sect_strt_cnt - 1

WARNING: Resource sharing could be useful in expression:
sect_strt_cnt := sect_strt_cnt - 1 and sect_strt_cnt := sect_strt_cnt - 1

WARNING: Resource sharing could be useful in expression:
sect_strt_cnt := sect_strt_cnt - 1 and sect_strt_cnt := sect_strt_cnt - 1

Development of an emulation flow
IF (load_byte_cnt = '1') THEN
    sect_byte_cnt := To_Integer( drive(section_length) ) + 3;
ELSIF (strtPriv_dat = '1') AND (pes_flt_act = '1') THEN
    sect_byte_cnt := To_Integer( drive(private_length) ) + 1;
ELSIF (sect_byte_cnt /= -1) THEN
    sect_byte_cnt := sect_byte_cnt - 1;
END IF;

WARNING: Resource sharing could be useful in expression:
sect_byte_cnt := To_Integer( drive(section_length) ) + 3 and
sect_byte_cnt := To_Integer( drive(private_length) ) + 1

IF (Pwr_Up_Rst /= '0') THEN -- checked this if statement
    ts_af_det_state := af_length;
ELSIF (strt_af_parser = '1') THEN
    ts_af_det_state := flags;
ELSIF (counted_bytes /= 10) THEN
    ts_af_det_state := prg_clk_ref;
ELSE
    IF (opcr_flag = '1') THEN -- checked this if statement
        ts_af_det_state := org_prg_clk_ref;
    ELSIF (spl_pnt_flag = '1') THEN
        ts_af_det_state := splice_countdown;
    ELSIF (ts_priv_flag = '1') THEN
        ts_af_det_state := ts_priv_data;
        length_fld_ind := '1';
    ELSIF (af_ext_flag = '1') THEN
        ts_af_det_state := af_extension;
        length_fld_ind := '1';
    ELSIF (counted_bytes = af_end) THEN
        ts_af_det_state := af_length;
    ELSE
        ts_af_det_state := af_stuffing;
    END IF;
END IF;

WARNING: Resource sharing could be useful in expression:
condition : (Pwr_Up_Rst /= '0') and condition : (counted_bytes /= 10)
WARNING: Resource sharing could be useful in expression:

condition : (Pwr_Up_Rst /= '0') and condition: (counted_bytes /= 10) AND (counted_bytes /= 16)
WARNING: Resource sharing could be useful in expression:
  condition: (Pwr_Up_Rst /= '0') and condition: (counted_bytes /= ts_priv_data_end)

CHECKING TEMPLATE OF STATE MACHINE IN psi_filter.vhdl.syn.vhdl

Development of an emulation flow
Is buffer_state the state variable of the declared state machine? (y/n)

y

CHECKING STATE ASSIGNMENTS OF STATE buffer_state

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>WHEN</td>
<td>wr =&gt;</td>
</tr>
<tr>
<td></td>
<td>buffer_state := rd;</td>
</tr>
<tr>
<td>WHEN</td>
<td>rd =&gt;</td>
</tr>
<tr>
<td></td>
<td>buffer_state := wr;</td>
</tr>
<tr>
<td>END CASE</td>
<td></td>
</tr>
</tbody>
</table>

WARNING: It could be possible that state buffer_state is not completely specified because not all CASE statements contain WHEN OTHERS clause.

CHECKING STATE ASSIGNMENTS OF STATE ts_af_det_state

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>WHEN</td>
<td>af_length =&gt;</td>
</tr>
<tr>
<td></td>
<td>length_fld_ind := '0';</td>
</tr>
<tr>
<td>IF</td>
<td>(Pwr_Up_Rst/= '0') THEN</td>
</tr>
<tr>
<td></td>
<td>ts_af_det_state := af_length;</td>
</tr>
<tr>
<td>ELSIF</td>
<td>(strt_af_parser = '1') THEN</td>
</tr>
<tr>
<td></td>
<td>tsAfDetState := flags;</td>
</tr>
<tr>
<td>ELSE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NULL;</td>
</tr>
<tr>
<td>END IF</td>
<td></td>
</tr>
<tr>
<td>WHEN</td>
<td>flags =&gt;</td>
</tr>
</tbody>
</table>

WARNING: State ts_af_det_state is not set in previous when-statement

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-- Process : pes_af_flt --</td>
<td></td>
</tr>
<tr>
<td>-- Description : --</td>
<td></td>
</tr>
<tr>
<td>-- Side Effects : --</td>
<td></td>
</tr>
<tr>
<td>pesAfFlt : PROCESS</td>
<td></td>
</tr>
<tr>
<td>TYPE lut IS ARRAY (0 TO 127) OF integer RANGE 9 TO 32;</td>
<td></td>
</tr>
<tr>
<td>VARIABLE prevAfState : tsAf_fsm_state;</td>
<td></td>
</tr>
<tr>
<td>VARIABLE pes_byte_cnt : integer RANGE 0 TO 33;</td>
<td></td>
</tr>
<tr>
<td>VARIABLE strt_writing0 : std_ulogic;</td>
<td></td>
</tr>
<tr>
<td>VARIABLE strt_writing1 : std_ulogic;</td>
<td></td>
</tr>
</tbody>
</table>
| VARIABLE pes_lut : lut := (9, 11, 10, 12, 10, 12, 11, 13, 12, 14, 13, 15, 13, 15, 14, 16, 15, 17, 16, 18, 16, 18, 17, 19, 18, 20, 19, 21, 19, 21, 20, 22, 9, 11, 10, 12, 10, 12, 11, 13, 12, 14, 13, 15, 13, 15, 14, 16, 15, 17, 16, 18, 16, 18, 17, 19, 18, 20, 19, 21, 19, 21, 20, 22, 14, 16, 15, 17, 15, 17, 16, 18, 17, 19, 18, 20, 18, 20, 19, 21,
TXT

BEGIN

-- replace of rising edge;

Is a reset declaration defined in entity (y/n) ?

n

ERROR: Initialisation of variables in a process is only permitted if a reset is used for this purpose. Synthesis Tools can’t guarantee initial other than through a reset signal, so a reset signal must be added.