Emulation flow for designs with large memory requirements

Lammers, K.J.

Award date:
1997

Master's Thesis

Emulation flow for designs with large memory requirements

K.J. Lammers

Department	: Design Automation Section, Department of Electrical Engineering, Eindhoven University Of Technology
Supervision	: Prof. Dr. Ing. J.A.G. Jess
Period	: 01-04-'93 to 18-02-'94
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.	: 270493

Abstract

During the development of HD-MAC decoder ICs for HDTV, an ASIC design flow was developed: the PCALE Design Flow. Since this design flow does not capture all aspects of system design, a *system* design flow is developed: the Advanced PCALE Design Flow.

Part of the Advanced PCALE Design Flow is *emulation*: the bread board implementation of a design through mapping of the VHDL description of the design to programmable logic devices. One of the steps for deriving this implementation is synthesis by means of synthesis tools. However, synthesis of designs containing large registers with programmable logic devices as back end turns out to be problematic, since very little memory is available in programmable logic devices. This large register problem was solved by using a RAM for implementation of a large register.

The solution consists of a conversion of the VHDL description of a design containing a large register to a VHDL description of a design containing a RAM while preserving design functionality. This conversion is feasible under certain restrictions and a tool was written to automate the conversion. Also, templates that guarantee that the restrictions are met, have been devised. Template checking is incorporated in the tool, prior to design conversion.

So through register replacement emulation of designs with large memory requirements has become possible within the Advanced PCALE Design Flow.

Summary

During the development of HD-MAC decoder ICs for HDTV, an ASIC design flow was developed: the PCALE Design Flow. Since this design flow does not capture all aspects of system design, a *system* design flow is developed: the Advanced PCALE Design Flow.

One of the new parts in the Advanced 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.The reasons for emulation are fourfold:

1. **Fast-prototyping**
2. **Start-up production**
3. **Field-test**
4. **Real-time simulation**

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.

One of the steps for deriving bread boards is synthesis by means of synthesis tools. However, synthesis of designs containing large registers with programmable logic devices as back end turns out to be problematic, since very little memory is available in programmable logic devices. This large register problem was solved by using a RAM for implementation of a large register.

Two solutions have been investigated, namely:

1. **Synthesis libraries**

   The three synthesis tools available at PCALE are reviewed with respect to their ability to add designer defined VHDL descriptions as new building blocks to their synthesis libraries. If it is possible to add a VHDL description of an existing memory IC to the synthesis libraries, then it might be possible to instruct the synthesis tool to automatically use this description instead of synthesizing the register.

   However, none of the reviewed synthesis tools support VHDL models as a basis for building blocks. The support that exists is not sufficient for application within the Advanced PCALE Design Flow. Therefore the conclusion is drawn that synthesis libraries cannot solve the large register problem.

2. **Register replacement**

   This solution consists of a conversion of the VHDL description of a design containing a large register to a VHDL description of a design containing a RAM while preserving design functionality. As it turns out, this conversion is feasible under certain restrictions.

Hence register replacement is the developed solution to the large register problem. A tool was written to automate the conversion. Also, templates that guarantee that the restrictions are met, have been devised. Template checking is incorporated in the tool, prior to design conversion.

So through register replacement emulation of designs with large memory requirements has become possible within the Advanced PCALE Design Flow.

Table of Contents

1. Introduction

When developing new systems, it is necessary to verify their performance prior to implementation 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 software 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 hardware only. For instance, most systems consist of both hardware and software. Also missing 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. Furthermore, 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 prototype 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.

A 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 development of a standard emulation flow is the subject of the Master's Thesis of L.P.M. van Lieshout (see [16]). He encountered several synthesis problems in the synthesis tools that can be used in the emulation flow for synthesizing a VHDL design description. Most of these problems were solved by defining a VHDL subset that is included in the (synthesizable) tool supported VHDL subset. But this subset could not solve all synthesis problems. One synthesis problem, the problem of large registers in the description of a design, still remained.

The reason that this synthesis problem cannot be solved by defining a VHDL subset is that this problem is not a question of "bad" VHDL statements but merely a question of register size. The size of a register becomes a problem when it exceeds the memory capacity of an EPLD: preferably the design is mapped into one EPLD and when more EPLDs have to be used, it is desirable in terms of timing not to divide the register amongst several EPLDs. Since the memory capacity of an EPLD is in the order of magnitude of 200 flip-flops, any register of that size or larger is a candidate problem register.

Therefore another solution has to be found for this synthesis problem. Hence the subject of this Master's Thesis, the development of an emulation flow for designs with large memory requirements.

Basic idea behind this emulation flow is the use of an existing memory IC (RAM) for the implementation of the large register. Then the large register no longer has to be implemented in flexible hardware; only the rest of the design has to be implemented in flexible hardware. This way the need for large memory capacity in flexible hardware would be taken care of.

The only way to solve the large register problem is to change the input for the synthesis tool, since the synthesis tool itself cannot be altered. The input for a synthesis tool are the description of the design that is synthesized and the synthesis libraries. Synthesis libraries provide the synthesis tool with the building blocks needed to synthesize the design. Both inputs are investigated in this Master's Thesis as possible solutions:

- First the possibilities to change the synthesis libraries of synthesis tools are evaluated. The three synthesis tools available at PCALE are reviewed with respect to their ability to add designer defined VHDL descriptions as new building blocks to their synthesis libraries. If it is possible to add a VHDL description of an existing memory IC to the synthesis libraries, it might be possible to instruct the synthesis tool to automatically use this description instead of synthesizing the register. Since the memory IC does not have to be synthesized (remember that it is an existing IC), only the rest of the design has to be implemented in flexible hardware.
- Secondly, the possibilities to change the VHDL description of the design are evaluated. It might be possible to change the VHDL description of a design in such a way that the register is replaced by (a VHDL model of) a memory IC, while preserving design functionality. In that case the standard emulation flow would have to be adapted in a way that the design itself is synthesized while the memory IC is not synthesized. The memory requirements imposed by the register are then no longer a problem. Of course the conditions under which the replacement can take place have to be investigated and preferably a tool is developed for this replacement.

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 standard emulation flow. Chapters 3 and 4 involve the development of the emulation flow for designs with large memory requirements itself: chapter 3 is about three synthesis tools and their synthesis libraries and chapter 4 reviews the conversion of a design with a register to a design with a memory IC. The templates that guarantee successful register replacement are discussed in chapter 5. Readers unfamiliar with the syntax and semantics of VHDL may experience some difficulties reading chapters 4 and 5. They are referred to the IEEE Standard VHDL Language Reference Manual (see [2]). The testcase and the tests that have been performed are the subjects of chapters 6 and 7. The features of the tool that performs the register replacement are also reviewed in chapter 8. The final chapter is concerned with conclusions regarding the development of an emulation flow for designs with large memory requirements.

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*

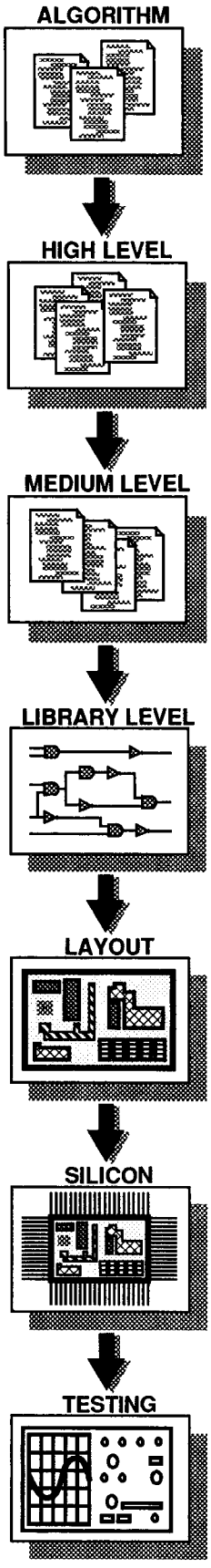
  A direct result of a short throughput time is a *short Time-To-Market.*

- 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 has been 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.

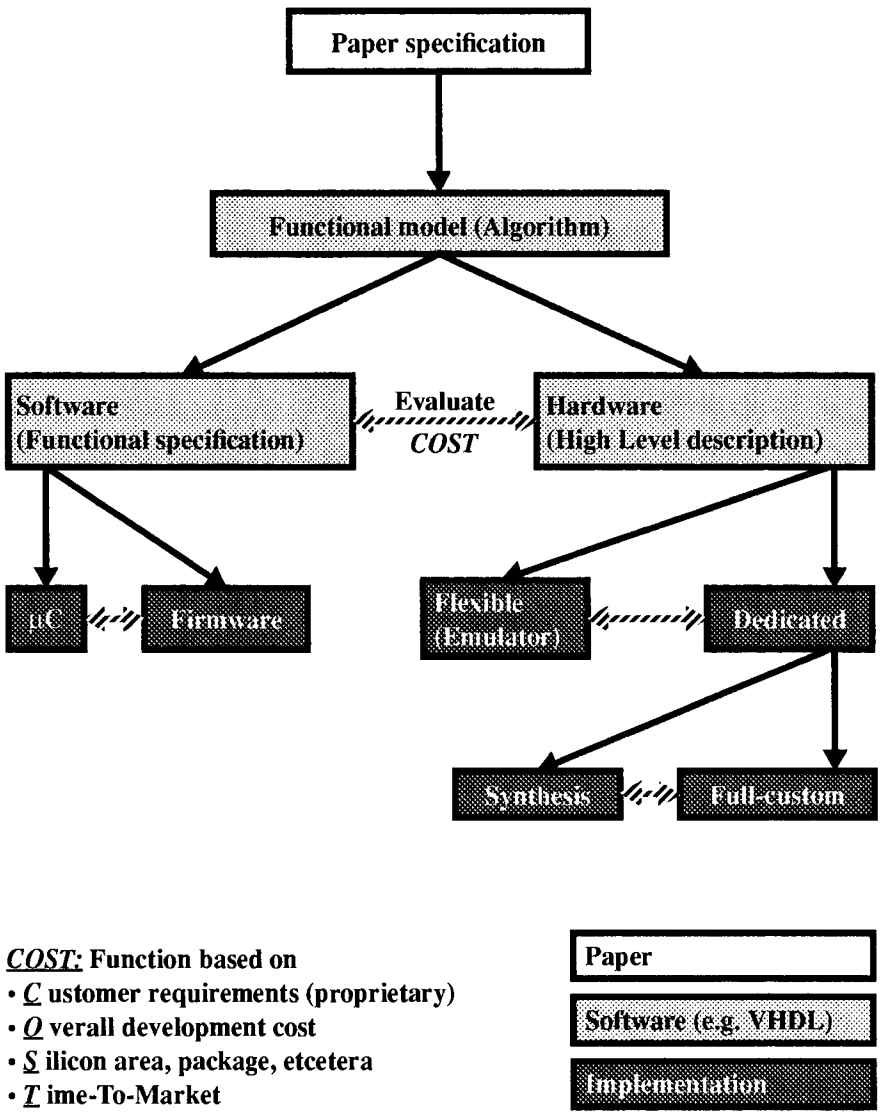


FIGURE 2. Advanced PCALE Design Flow

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 micro-controller (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 in general has been developed by L.P.M. van Lieshout. However, designs with large memory requirements cannot be emulated with that flexible hardware route. Hence this Master's Thesis subject of a flexible hardware route for such designs.

- **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 established. 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.

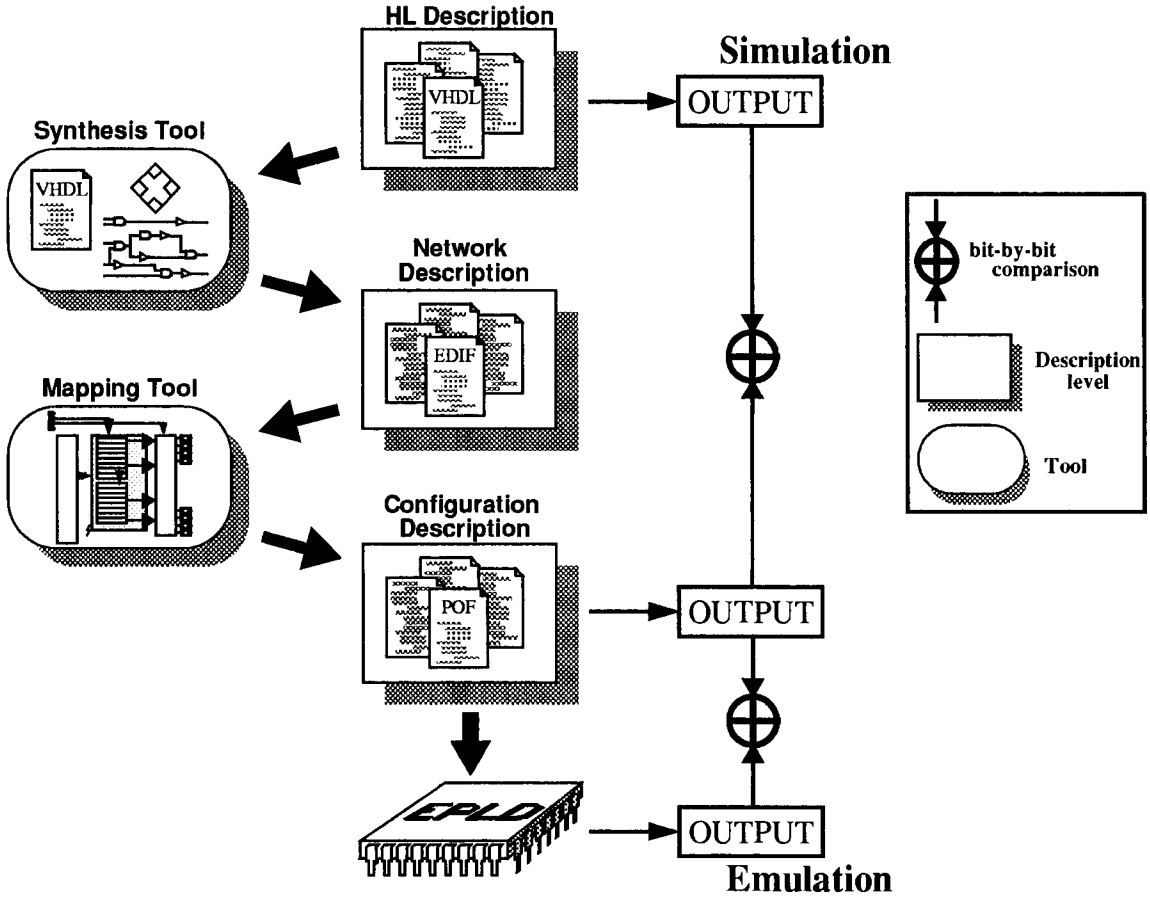


FIGURE 3. Concept emulation flow

However, the synthesis of a design by means of a synthesis tool turns out to be the most critical step in the emulation flow due to the fact that the available synthesis tools demonstrate several problems when synthesizing an HL description written in VHDL. These problems have been thoroughly described by L.P.M. van Lieshout in his Master's Thesis (see [16]). He describes various subsets of the VHDL language (see figure 4) and concludes that many of the synthesis problems stem from the fact that these subsets have non-overlapping parts and that even VHDL descriptions written according to the tool supported subset are possibly synthesized inefficiently.

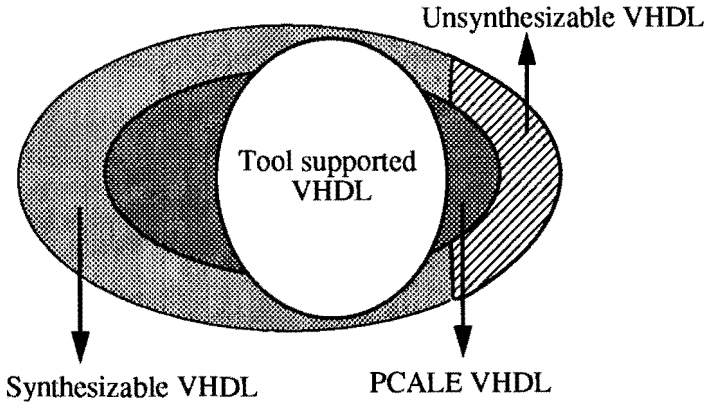


FIGURE 4. VHDL subsets

As a solution to the synthesis problems he defines the Design Style Assistant (DSA) VHDL subset, a VHDL subset that contains those VHDL statements of the tool supported VHDL subset that are synthesized in a satisfactory way. Figure 5 illustrates the position of the DSA VHDL subset with respect to the other VHDL subsets.

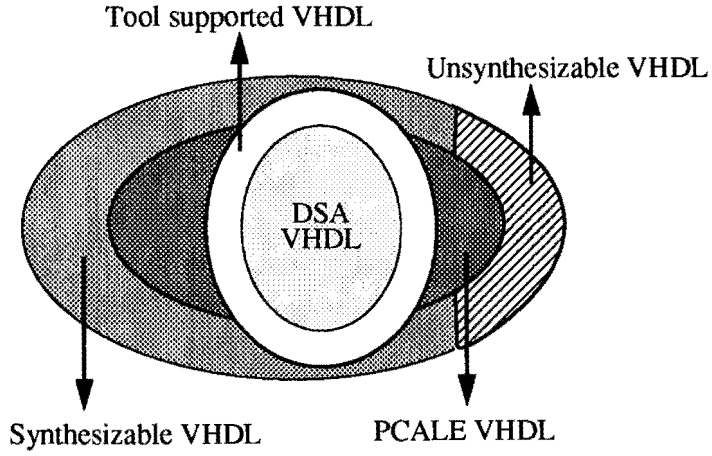


FIGURE 5. DSA VHDL subset

By making sure that an HL description not only complies with the PCALE design VHDL subset but also with the DSA VHDL subset, most of the synthesis problems can be solved except for the large register problem, since this problem is not a question of "bad" VHDL statements (registers are allowed within the DSA VHDL subset) but merely a question of memory size.

A DSA tool has been developed by L.P.M. van Lieshout to help the designer to ensure that an HL complies with the DSA VHDL subset. This DSA tool can be used to aid in translating VHDL constructs in the PCALE design VHDL subset, but not in the DSA VHDL subset to equivalent VHDL constructs in the DSA VHDL subset. This DSA tool is to be applied before synthesis is attempted. The emulation flow incorporating this tool is depicted in figure 6.

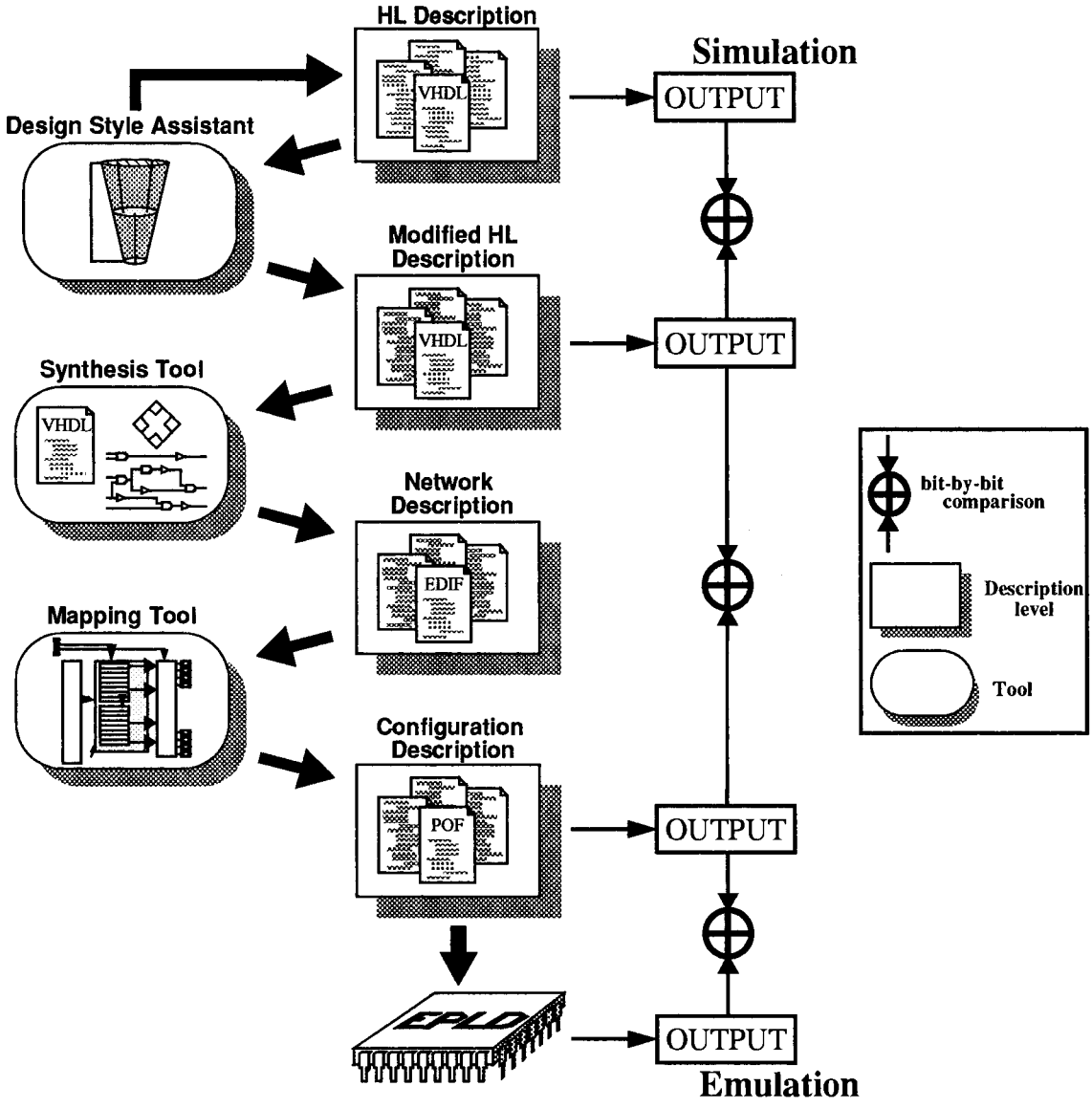


FIGURE 6. Standard emulation flow

This emulation flow is called the standard emulation flow since it applies to almost every design and concerns most of the synthesis problems. Whether or not this emulation flow also holds for designs with large registers, is to be investigated.

3. Synthesis libraries

When synthesizing an IC description, synthesis tools use so-called synthesis libraries to map the description to a technology. These synthesis libraries provide the synthesis tool with the building blocks and their characteristics needed to perform synthesis and to be able to meet constraints. Constraints may have been imposed on the design in order to achieve speed or memory requirements. For every technology another synthesis library is available since building blocks and their characteristics can differ from technology to technology.

When synthesis libraries can be expanded with designer defined building blocks, this provides a powerful way to perform more efficient mapping. Since the synthesis tool can choose between the original building blocks and added optimized building blocks, the mapping is likely to improve: inefficient mappings can be avoided and previously impossible mappings become available. In fact, effort that has previously been invested into certain designs, is not lost. Also, if existing ICs are added as building blocks, implementation of a building block by an existing IC and copying (parts of) layouts of existing ICs are possibilities that come within reach. So chip design becomes more efficient and faster.

But the fact that (parts of) designs can be mapped to added building blocks offers the possibility to emulate designs containing one or more large registers. If a RAM is added as building block to the synthesis libraries and if the synthesis tool can map a register to such a RAM, then the register can be implemented by a RAM instead of by flexible hardware. This way the emulation flow can be extended to cover designs with large registers.

In the next sections three synthesis tools are discussed. They are evaluated in terms of their ability to add designer defined building blocks to their synthesis libraries. The evaluation concerns designer defined building blocks in general. This is done because a general solution not only enables emulation of designs with large registers but also upgrades chip design. A solution merely aimed at the addition of a RAM as building block would only enable the emulation and would not improve chip design. Since all design descriptions at PCALE are written in VHDL, the synthesis tools are evaluated on their possibility to add VHDL descriptions of building blocks to their synthesis libraries; descriptions that are written in a format other than VHDL are not of interest. And, of course, all results must fit into and comply with the Advanced PCALE Design Flow and especially the Flexible Hardware route, since the synthesis tools are applied in that context.

3.1. Autologic

The synthesis tool from the Mentor Graphics Corporation is called Autologic. It is the first synthesis tool that has been evaluated with respect to its ability to add designer defined building blocks to its synthesis library. The manuals that go along with the synthesis tool (see [6] and [7]) form the basis of this evaluation.

An Autologic synthesis library can be developed through a library development process. This development process involves a number of steps, starting with the creation of a so-

called AMP-library. Only from such an AMP-library, Autologic synthesis libraries can be developed.

The only way to add a component to an AMP-library is to provide a functional model of that component. Three AMP functional model types are supported:

1. **AMP built-in model**

   An AMP built-in model is a logical model that operates with simulators and synthesis tools and provides good run time performance and high memory capacity in the tools.

2. **QuickPart table model**

   A QuickPart table model is a formatted table description of a functional model which offers specification flexibility for the creation of accurate models.

3. **Schematic model**

   A schematic model is a collection of smaller models (such as built-in models and QuickPart Table models) which accurately represents the internal composition of the model.

VHDL models, however, are not supported, so an alternate model must be provided for such VHDL models or the unsupported VHDL models must be marked as "blackbox".

A. Providing an alternate model for the VHDL model

This means that the designer has to supply an extra model for synthesis with the same functionality as the original VHDL model. This means that the same objective can be achieved with the alternate model; the VHDL model is superfluous. Moreover, "modelling the VHDL model" is not wanted: you are performing the same task, namely describing the functionality of a design, twice. The only remaining possibility is generating a schematic from the VHDL model. Then this schematic could be used as a functional model for the AMP-library. However, schematics that can be used as a functional model for the AMP-library are restricted:

- Schematics can only consist of built-in models, QuickPart Table models or other schematic models. VHDL models cannot be part of schematics.
- Asynchronous feedback cannot be included in a schematic model.
- Restrictions are present for the components that can and/or should be used in a schematic model. That is: some components are not supported and some components are mapped in a logically correct, but inefficient manner. Since there is no guarantee that such unwanted components are introduced when generating the schematic, there is also no guarantee that the schematic can be used as a functional model for the AMP-library.

Looking at the above restrictions, it is evident that no guarantee can be given in advance whether or not a schematic generated from a VHDL model can be used as a functional model for the AMP-library. Therefore schematics do not qualify as a possible solution.

B. Marking the VHDL model as blackbox

Marking the VHDL model as blackbox means that no logic is generated during synthesis. The designer or the library developer must create a replacement rule to map the blackbox cell to a target library. By means of a replacement rule Autologic can be instructed to use a specific building block from the library for implementation of a certain component.

In case of VHDL models of existing ICs, no optimization should be performed, so marking of the blackbox cell with Syn_donttouch or Syn_dontuse is necessary.

- Syn_donttouch: Autologic does not remove or introduce this component during optimization, but it can still be replaced by means of a replacement rule.
- Syn_dontuse: Autologic does not introduce this cell during optimization, but it can be introduced by means of a replacement rule.

So marking the VHDL model as blackbox and replacing it through a replacement rule boils down to instantiating a netlist from the synthesis library. Since timing is not included in the netlist, the timing of the total IC can only be synthesized and optimized correctly if the designer tells Autologic what the timing of the blackbox is. On top of that, in the case of EPLDs as back end synthesis tools cannot guarantee that the timing will be correct after synthesis and mapping.

This is not a straightforward and transparent use of synthesis libraries since nothing is done by the synthesis tool: what to replace, how to replace it and the timing have to be entered by the designer. So marking the VHDL model as blackbox is useless.

Since there is no way to use VHDL models in the envisaged fashion as a basis for components of the synthesis libraries of Autologic, the Autologic synthesis libraries cannot solve the problem of large registers in a design.

3.2. CORE

Another synthesis tool is the Complete Optimization and Retargeting Environment (CORE) from Exemplar Logic. It is the second synthesis tool that was reviewed. In CORE there is a distinction between input synthesis libraries and output synthesis libraries. Figure 7 illustrates this distinction.

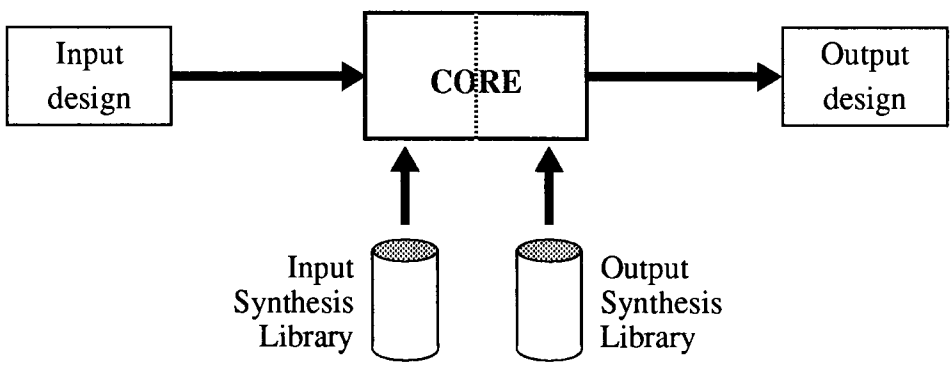


FIGURE 7. CORE and synthesis libraries

CORE uses input synthesis libraries to allow mapping out of a technology, that is to provide an input for CORE. Output synthesis libraries are used for mapping into a technology.

Concerning synthesis libraries, CORE has two tools available for the creation of synthesis libraries: *lBuild* for the creation of libraries that are used for input synthesis only and *lGen* for the creation of libraries that are used for input and output synthesis. The restrictions of these tools and particularly the restrictions on the synthesis libraries are listed below. Note that on top of these restrictions, every library element has to be described by hand. Automatic creation of library elements is not supported so VHDL models cannot be automatically transformed to library elements.

lBuild:

If a library is built for input synthesis only, the area and electrical properties of the gates are not needed, and the global library properties are also not needed. Only gate functionality descriptions are required. Gate functionality can be described as a set of boolean equations, as a set of predefined primitives or a combination of both. The following restrictions for input synthesis libraries are listed in the *lBuild* manual (see [9]):

- Supply voltage with the name VCC and ground with the name GND are added when the synthesis library is built, unless those functions are specified with other names.
- Gate and pin names are case insensitive.
- No loops in combinatorial logic are allowed. Most loop situations can be specified by using one of the predefined primitives.
- Gate, pin and node names which are also keywords or contain non-alphanumeric characters, should be quoted. For example: gate "DELAY".

lGen:

lGen is the tool to use for libraries that are built for output synthesis also.

In addition to the restrictions for input synthesis libraries listed in the *lBuild* manual, the *lGen* manual (see [10]) lists the following restrictions for output synthesis libraries:

- *lGen* requires at least a 2-input NAND-gate and an inverter to be included in the library.
- Complex combinatorial cells with more than one output are allowed for input only. Therefore these cells must be designated as NOMAP when the library is used for output synthesis also.
- For mappable gates, *lGen* supports at most a single predefined primitive (such as flip-flops, latches, tri-states) per gate. There is no limitation to the combinatorial logic that can accompany that primitive.
- When mapping into a technology, CORE performs automatic selection of gates in a class to get the best performance out of the circuit. Gates are members of the same class if they have the same functionality and the same pin names.

The synthesis libraries are used as output synthesis libraries in the PCALE Design Flow. Since it is far from realistic that library elements (and in particular RAMs) contain at the most one primitive and have only one output, and since VHDL models cannot be used as a basis for library elements, the CORE synthesis tool is unfit as a solution to the memory problem that large registers cause in the flexible hardware route.

3.3. VHDLSyn

The last synthesis tool that was evaluated, is VHDLSyn from Philips Electronic Design & Tools. According to the VHDLSyn manual (see [11]), it is possible to add your own parametrized VHDL models to the synthesis library. Since VHDLSyn can produce output in the Sprite Input Language (SIL) format, the SIL-format of the library is not a problem.

VHDLSyn uses an implementation from the synthesis library whenever a VHDL operation is translated or when an entity or subprogram is instantiated that has the so-called *lv_primitive* attribute (see [11]). This means that by giving an entity or a subprogram the *lv_primitive* attribute, VHDLSyn can be told to skip the architecture/subprogram body and to use a predefined implementation from the synthesis library.

Automatic selection of an implementation is only possible for standard building blocks (such as AND-gates, flip-flops, etcetera) or for standard operations (such as counters, shifters, etcetera). This means that you can add your own components to the synthesis library without any problem, but the automatic selection only takes place if the added component is an alternative for already existing standard building blocks or standard operations.

Combining the memory problem with the possibilities of VHDLSyn, two problems are noticed:

1. RAMs and registers are completely different hardware components. A RAM does not qualify as an alternative for a register. That makes automatic selection impossible in VHDLSyn.
2. A register can be modelled in VHDL as a VARIABLE or as a SIGNAL. VARIABLES and SIGNALS are *part* of an entity or subprogram. So the *lv_primitive* attribute is also useless since that attribute can only be applied to entities or subprograms as a whole.

Apparently VHDLSyn does not incorporate capabilities that allow automatic implementation of a register by a RAM. Therefore it is evident that the memory problem cannot be solved by the synthesis libraries of VHDLSyn.

3.4. Conclusions

A large register in a design requires a large memory capacity to be available in flexible hardware when a bread board is developed for the design. Flexible hardware elements in contrast have very little memory capacity. Therefore synthesis tools have to use a lot of

flexible hardware elements when mapping a design which contains large registers to flexible hardware. This means that a design has to be partitioned among multiple elements.

Partitioning implies an increase in wiring and in wiring complexity, which leads to inefficient mappings, which in turn cause a decrease in clock frequency. Remember that for video applications the speed requirements are very stringent. 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 this, the synthesis tools crash during synthesis of such designs, probably because such large registers cause an overflow in the internal format used by the synthesis tools.

To overcome the memory problem, several synthesis tools have been reviewed on their synthesis libraries. The idea behind the use of synthesis libraries is quite simple: if it is possible to add a VHDL model of a RAM to these libraries as new building block, then it might be possible that the synthesis tool can perform an automatic mapping of the register to this new building block.

But the reviewed synthesis tools have either no support for VHDL models as a basis for their synthesis libraries, or such support has too many and too severe restrictions. Hence none of the reviewed synthesis tools can provide a solution to the memory problem in terms of synthesis libraries.

4. Changing the design description

Besides the synthesis libraries of synthesis tools, another solution is looked into: perhaps the description of a design containing a large register can be altered in such a way that the large register problem can be overcome. The basic idea is still that the large register is implemented in an existing RAM IC while the rest of the design is implemented in flexible hardware. The bread board for emulation can thus be built from this RAM and flexible hardware. The emulation flow describing the approach of this solution is shown in figure 8.

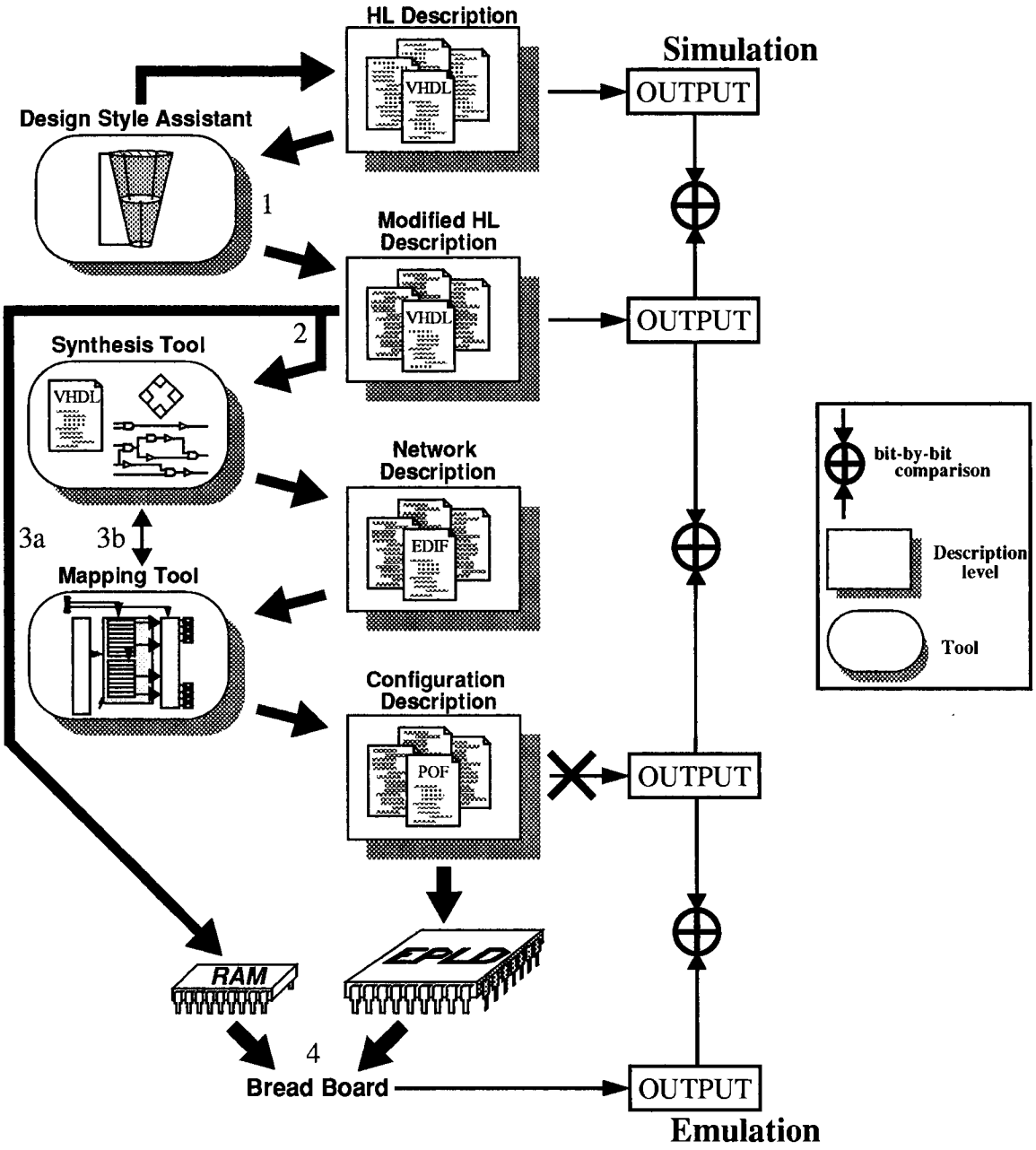


FIGURE 8. Emulation flow for designs with large memory requirements

In this emulation flow the consecutive steps to take, starting with an HL description of a design, are:

1. **Conversion of the design description**

   The first step in the emulation flow is the conversion of the HL description of the design to a new HL description. During this conversion the large register has to be completely replaced by a RAM. The tool that performs the replacement is part of the Design Style Assistant (DSA) tool that already was present in the standard emulation flow of figure 6.

   The new description still has to be simulatable of course for functional verification of the design after conversion. Recall that functional verification is an essential part of the PCALE Design Flow and that the emulation flow must comply with the PCALE Design Flow.

   However, the new design description does not have to be synthesizable. The RAM, actually a VHDL model of the RAM, only has to be simulatable since it is not synthesized. The rest of the design description on the other hand must be completely synthesizable since it has to be synthesized by means of a synthesis tool.

2. **Splitting the design description**

   The second step is splitting up the new HL description into a part containing the RAM and a part containing the rest of the design. This has to be done since the RAM is not synthesized.

3a. Acquisition of a suitable RAM

Evidently, a suitable RAM must be acquired for implementation of the register unless it is available. A suitable RAM is a RAM that suffices the speed, memory and voltage requirements.

3b. Synthesis and mapping

Parallel to the acquisition of a suitable RAM is synthesis. Synthesis is performed on the new HL description without the part that describes the RAM. After synthesis, mapping on flexible hardware (EPLDs) is performed by means of a mapping tool.

4. **Bread board building**

   After RAM acquisition and synthesis, it is time to build the bread board. On this bread board the flexible hardware and the RAM are connected. Finally the bread board is emulated and the emulation results are verified through bit-by-bit comparison with the simulation results that are obtained after conversion of the design description.

The first two steps in this emulation flow still have to be developed; the other steps can already be performed. But before these two steps can be developed, the differences between registers and RAMs must be inventorised. Since RAMs and registers differ in behaviour, it is very likely that there are some restrictions on the conversion of the design description. After these restrictions have been determined, conversion and splitting of the design can be developed, thus completing the emulation flow for designs with large memory requirements.

4.1. Differences between RAMs and registers

For the replacement of a register by a RAM the most general case is considered: registers with full random access and assignment. Afterwards it is always possible to investigate the replacement of other registers such as shift registers. Perhaps the restrictions for replacement and the replacement itself are different for such other registers.

Preferably there are no restrictions for the register replacement, of course. This means that full random register access and assignment have to be replaced by full random RAM access and assignment. But that is an illusion since the behaviour of a RAM is very different from the behaviour of a register. Due to the differences between RAMs and registers there are most likely some restrictions for describing a register in a design when such a register is to be replaced by a RAM. The main differences between RAMs and registers are:

1. **Response time**

   Every operation on a RAM takes a certain amount of time namely the response time of that RAM. A register in contrast has no response time. The time needed to perform a register operation is negligible compared with the time needed for a RAM operation.

2. **Accessible amount of data**

   Only one word at a time can be read from a RAM or written to a RAM. This means that for a RAM the accessible amount of data is equal to the wordlength of the words of the RAM. The accessible amount of data for the register on the other hand is equal to the registersize since the register is completely accessible.

3. **Control signals**

   Besides the clock signal the register does not need any control signals for its operation. A RAM however does not need a clock signal but some other control signals, namely a read_write signal, an enable signal and an acknowledge signal.

   The read_write signal indicates the RAM what kind of operation it has to perform: a read operation or a write operation. With the enabling signal the RAM can be turned on and off. Usually turning the RAM off results in a low power consumption. The RAM has to be enabled before data is read from the RAM or written to the RAM. The third control signal, the acknowledge signal, is used by the RAM to indicate that an operation is completed.

These differences clearly indicate that the conversion of the HL description is not trivial. But besides the differences between registers and RAMs, there are more aspects involved in the conversion. For instance, the HL description has to be simulatable after the conversion and the design without the RAM has to be synthesizable.

The complexity of the conversion is depicted in figures 9 and 10. In figure 9 the position of the design in the various VHDL subsets before replacement is reflected.

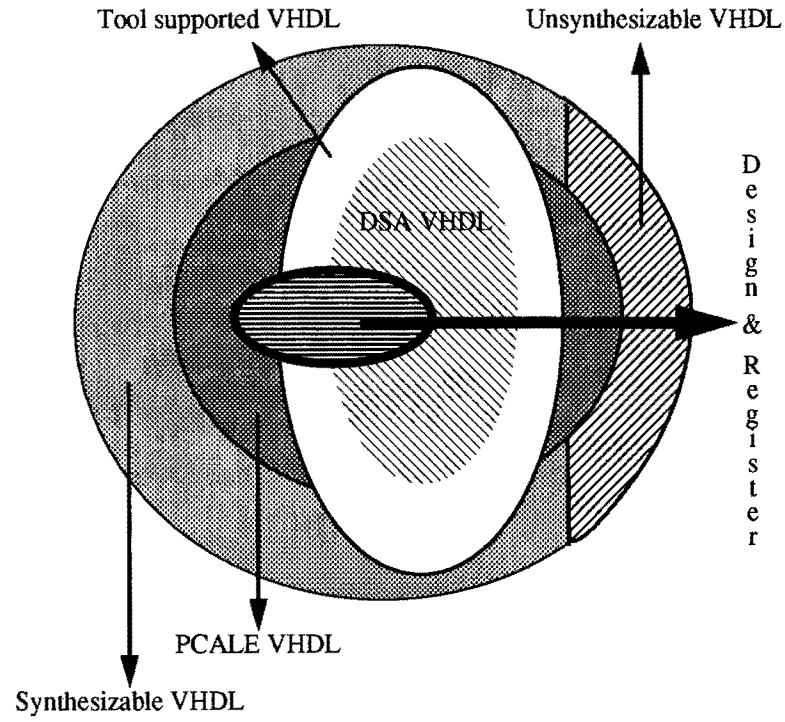


FIGURE 9. Place of design description in VHDL subsets before conversion

Figure 10 shows the position of the design with the RAM in the VHDL subsets after the replacement.

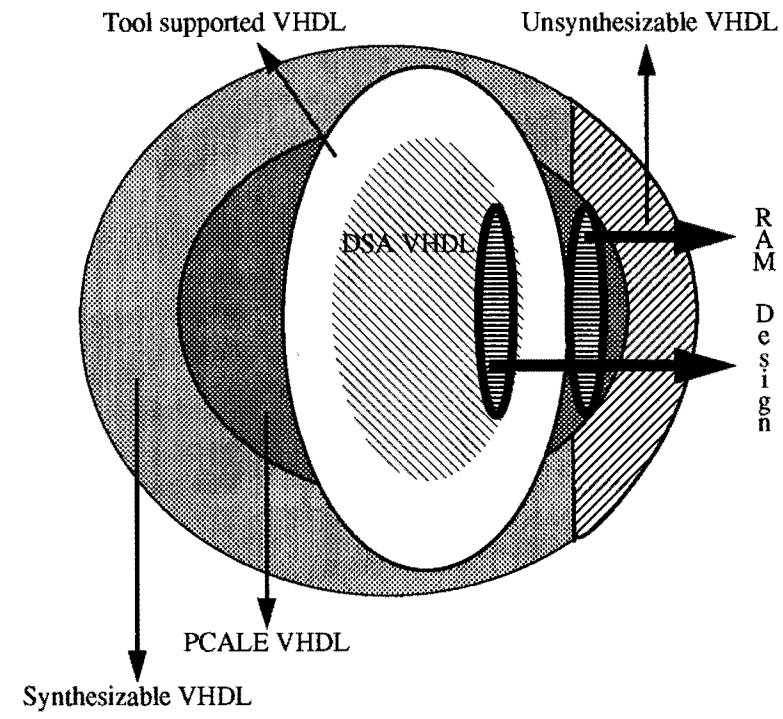


FIGURE 10. Place of design description in VHDL subsets after conversion

As figure 10 clearly shows, the RAM is in the PCALE VHDL subset but is not synthesizable. The rest of the design is in the DSA VHDL subset after treatment by the DSA tool.

4.2. Replacement restrictions

As stated in the previous section it is most likely that the replacement of a register by a RAM is restricted. Before the restrictions for replacement can be stated there are a number of considerations that must be paid attention to. These considerations that are listed below lead to the restrictions under which the replacement can take place and thus they lead to the VHDL model of the RAM.

Considerations:

1. A RAM consists of two parts: the memory itself and an addressgenerator which is used to address the RAM correctly. This is illustrated in figure 11. Also indicated in the figure are the signals that control the memory and the addressgenerator.

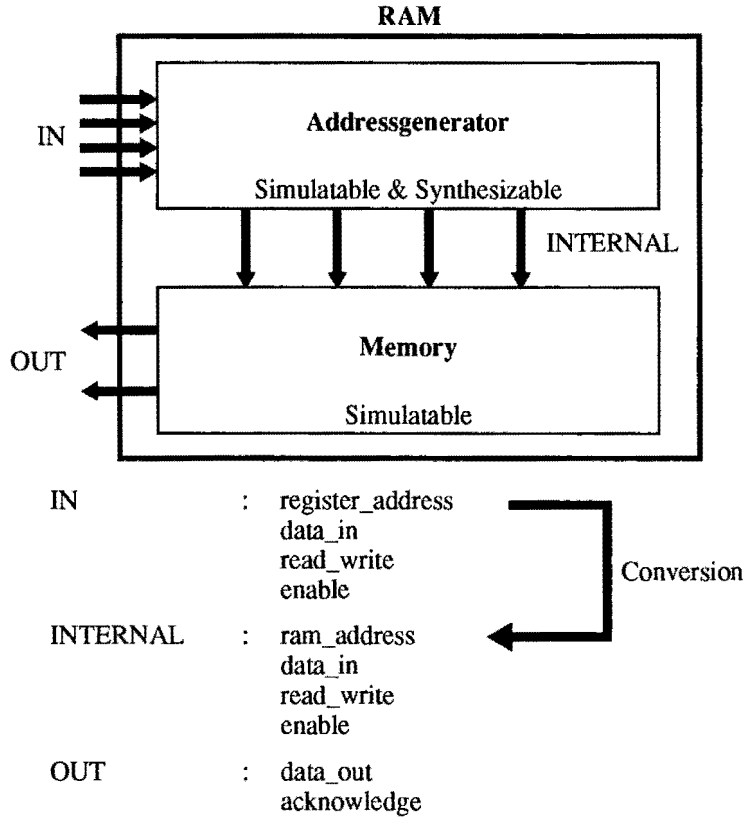


FIGURE 11. Schematic model of a RAM

The memory consists of a certain number of words, each with the same wordlength. The total memory capacity of the RAM is the wordlength multiplied with the number of words.

A register can be modelled as a one-dimensional and as a two-dimensional array as shown in figure 12. More dimensional arrays are not supported by synthesis tools. Therefore registers are assumed to be one-dimensional or two-dimensional since the result of a replacement of a more dimensional register cannot be verified.

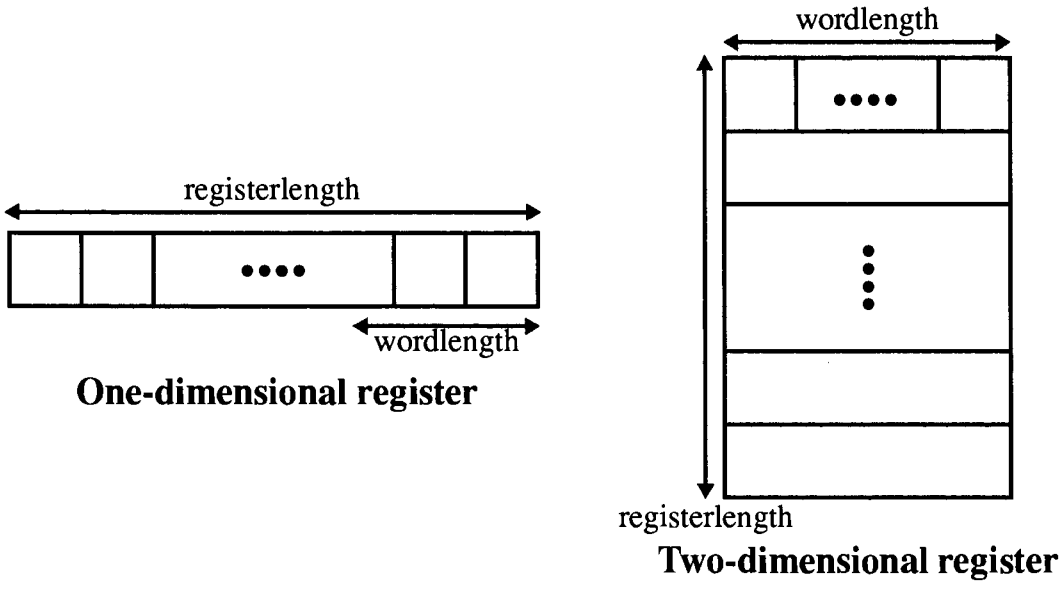


FIGURE 12. Schematic models of a one-dimensional and a two-dimensional register

Registers are indexed with integer values. In contrast, RAMs are addressed with so-called std_(u)logic_vectors. Hence type conversion is necessary. This type conversion has to take place in the addressgenerator. Note that the memory of a RAM is always two-dimensional. A different dimensionality between the register and the memory requires a more complex address conversion of course.

2. For the memory of the RAM an existing IC is used while on the other hand the addressgenerator has to be synthesized. Therefore it is necessary that they have separate VHDL models. The VHDL model of the *addressgenerator* must be both *simulatable and synthesizable* whereas the VHDL model of the *memory* only needs to be *simulatable*.

3. In the design various parts of the register (slices) can be indexed. It is possible that these slices vary in size or differ from the wordlength of the memory. This has to be accounted for in the addressgenerator also. For simplicity only *slices of a size equal to the wordlength* are considered (see figure 12). Afterwards extensions can be added.

4. Another consideration is that *constraints* have to be put on the various signals. Every RAM has its own specific response time: every read-operation and every write-operation takes this time to complete. This time can be accounted for in the simulatable model of the RAM memory; however, this time is not accounted for when synthesizing the design and the addressgenerator. Therefore constraints have to be put on various signals (such as the read_write signal, the data signal, etcetera), in order to synthesize a functionally correct design and addressgenerator. These constraints are parameters for the synthesis tool when it performs synthesis.

   There are two types of constraints: input constraints and output constraints:

   - With input constraints the arrival times at input ports can be defined. An arrival time for an input port defines the maximum delay relative to the clock to that input through logic external to the synthesized design.

- With output constraints the required times at output ports can be defined. The required time for an output port defines the longest allowable path from any input port to the output port. Paths start at primary inputs and at register outputs. Paths end at primary outputs and at register inputs.

Constraints are relative to the last or to the next active clock edge (see figures 13 and 14).

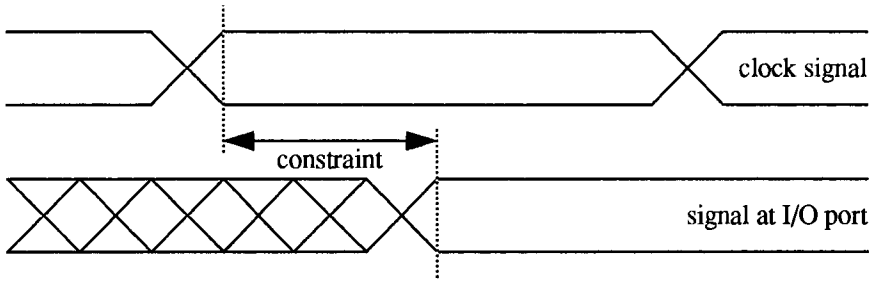


FIGURE 13. Constraint relative to last active clock edge

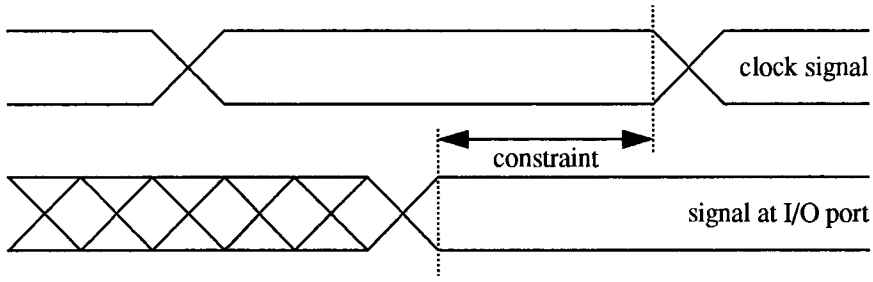


FIGURE 14. Constraint relative to next active clock edge

Note that when signals are constrained, there is no guarantee that the synthesis tool is able to synthesize a network that meets the constraints imposed on the input/output signals of the design and addressgenerator. Even if the synthesis tool reports successful synthesis, it is still not guaranteed that the constraints are indeed met, since the exact timing of the final EPLD(s) is not known before the mapping is completed (see [16]).

5. In VHDL the signals that control the RAM can be modelled as VARIABLES or as SIGNALS. Preferably the RAM that replaces a large register has the same possibilities that the register has: full random access for write and for read operations. This implies the use of VARIABLES.

   It is already established that the signals have to be constrained. Since only SIGNALS can be constrained, the use of VARIABLES is impossible. A direct consequence of the use of SIGNALS is that *the number of operations per clock cycle is limited to the maximum of one.* This is due to the fact that SIGNALS can only be updated once per clock cycle in VHDL.

   Another consequence of the usage of SIGNALS is that *read operations have to be anticipated at least one clock cycle.* Suppose some data is needed in the *middle* of a PROCESS. That implies that the signals have to be updated in the *middle* of the PROCESS. On the contrary, during simulation SIGNALS are updated at the *end* of a PROCESS.

The only way in VHDL to update SIGNALS in the middle of a PROCESS is by using a WAIT-statement. However, WAIT-statements are not synthesizable. So when data is needed in a clock cycle, the data has to be retrieved before that clock cycle.

6. The replacement of a register by a RAM alone is complex enough, so the tool that eventually has to carry out the replacement does not try to ensure that the register complies with the restrictions for replacement. Instead, the restrictions under which the replacement can be done are captured in templates (see chapter 5) and the tool checks the design before replacement for compliance with these templates.

   If the templates are violated, the replacement is not guaranteed to be correct. In that case interaction with the designer rules the decision whether or not a register has to be replaced by a RAM. This means that the designer takes responsibility for the restriction that the register complies with the restrictions.

7. Another important aspect is the *timing*. The timing of a RAM is shown in the diagram of figure 15. Normally when addressing a RAM, all the needed signals (read_write, data, address) are made valid. After that the RAM is enabled by means of an enable signal. When the read or write operation has completed, the RAM sends an acknowledge signal to indicate this completion. Next the enable signal is disabled. In case of a read operation data is not sent to the RAM but data is received from the RAM, of course.

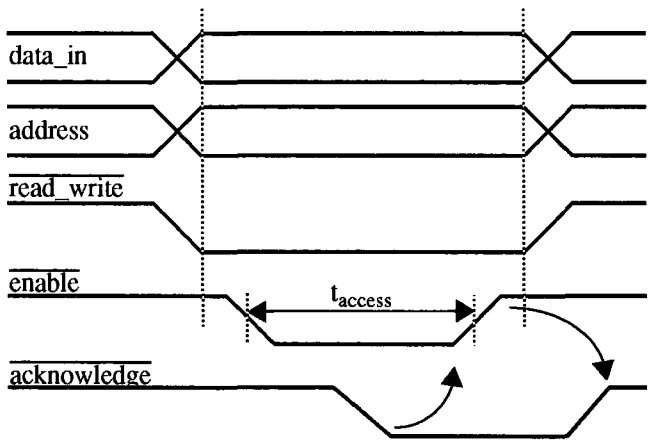


FIGURE 15. Timing diagram of a RAM

The VHDL code that corresponds with the above timing diagram and that handles the acknowledge correctly, is:

```
valid_data_in;     -- in case of a write operation
valid_address;
valid_read_write;

enable_ram;
WAIT UNTIL ram_acknowledge;
disable_ram;
```

Yet such a construct, to be implemented in the addressgenerator, is not synthesizable due to the fact that the WAIT UNTIL statement is not synthesizable for signals other than the clock signal (see [16]).

Another solution one might think of is to use a WAIT FOR statement in the VHDL model of the memory:

```
WAIT FOR access_time;
disable_ram;
```

Since the memory is not synthesized, the unsynthesizability of this statement is not a problem. But the enabling and disabling of the enable signal is a problem. Because the addressgenerator "knows" when the various signals are valid, the enabling has to be done by the addressgenerator. Disabling on the other hand is done by the memory, since only the memory can determine when the operation has been completed.

However, there are three reasons why this approach cannot be applied.

1. It is functionally incorrect that a RAM disables itself. The RAM is controlled by the addressgenerator so the addressgenerator has to enable and disable the RAM.
2. The logic needed to disable the RAM is not synthesized since the memory of the RAM is not synthesized.
3. The enable signal now has multiple drivers: both the memory and the addressgenerator assign the enable signal. Multiple drivers imply a WIRED_OR or a WIRED_AND in hardware and such hardware elements are not present in EPLDs. A separate process for disabling the RAM, triggered by an EVENT on the acknowledge signal, is also impossible due to multiple drivers.

The only remaining possibility is that an enable VARIABLE is used instead of an enable SIGNAL, but since constraints cannot be applied to VARIABLES, this cannot be correct either.

Apparently *the enable signal must always be enabled*, since there is no way of implementing correct disabling.

8. Since the RAM is always enabled, the RAM is continuously working. However, the design does not continuously access the RAM, and the data and/or address signal may change after an access. In case the last performed operation was a read operation, a change in the data and/or address signal does not matter. But if the last performed action was a write operation, any change in the data and/or address signal has a disastrous effect: the RAM contents are unintentionally overwritten. The only way to avoid this effect is to be *always reading the RAM, unless explicitly a write operation has to be carried out.* So after a write operation, the read_write signal is immediately adjusted so that the writing of the RAM only takes place when correct data and address signals are present.
9. Let us consider the read operation. Several cases can be distinguished for the read operation:
   - Only a read operation is performed.
   - The read operation is performed after certain conditions have been met. The determination of these conditions is implemented in hardware as a boolean network. This network evidently takes some amount of time before the read operation can start.

- Furthermore, it is possible that a decision is made on which data is read, whereas every read operation can be preceded by some decision. Since every decision implies another boolean network, it is most likely that the read-operations are preceded by different amounts of time.
- It is also not unlikely that a read operation is followed by some sort of calculation on the read data. This calculation also takes time in hardware and again different amounts of time can be expected.

Now two problems arise:

1. There is no way of predicting the amount of time a synthesized network (boolean network, network for calculation) takes.
2. Every signal (read_write, data, address, enable) can have only one constraint, while the preceding indicates that all the different cases ask for different constraints (meeting one constraint most likely violates other constraints).

To solve these problems, the constraints could be determined iteratively. This iterative determination starts by giving the various output signals an output constraint of 0 ns after the last active edge of the clock; the input signal (data coming from the RAM) is given an input constraint of 0 ns before the next active edge of the clock. In fact this means that the determination starts with the assumption that there is no preceding boolean network before a read operation and no following calculation after the read operation. The start of the constraint determination is depicted in figure 16.

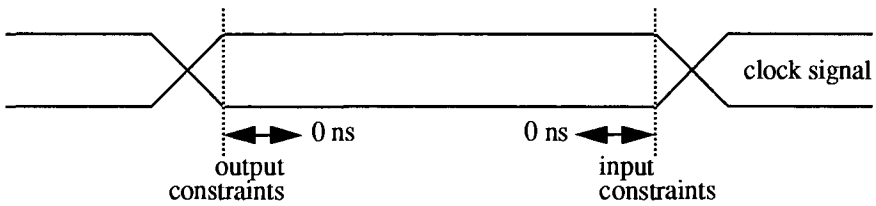


FIGURE 16. Start of iterative constraint determination

Starting with constraints of 0 ns, the synthesis tool could issue an error about the times minimally needed for the boolean network and for the calculation. These times are then the new constraints in the iterative process of constraint determination. The correct constraints can thus be found. Note that the time between the output constraints and the input constraints must be at least equal to the response time of the RAM, otherwise there is not enough time to perform an operation on the RAM. The end of the constraint determination is shown in figure 17.

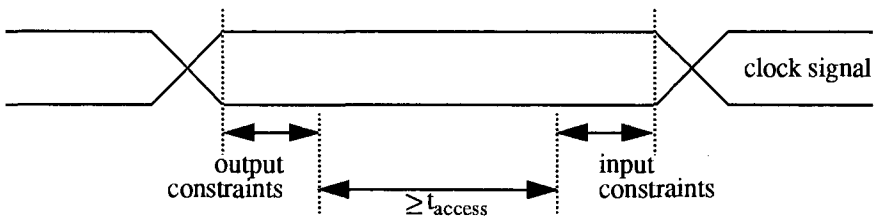


FIGURE 17. End of iterative constraint determination

It is not known whether there is a synthesis tool that reports such a warning. It is also unknown whether the synthesis tools report the time minimally needed for each and every read operation or only for the first read operation that cannot be successfully implemented with the given constraint. This could mean that the designer is *iteratively determining the constraints*, while in the end the conclusion might be that the needed input and output constraints cannot be met.

Note that the constraints on the signals for write operations coincide with the constraints for read operations; the constraints would have been adversary if the read operation is implemented in the same clock cycle when it is needed. Actually this is another reason for anticipating the read operations.

10. The memory part of the RAM has to modelled but is not synthesized. However, there must be an EVENT that triggers the VHDL model of the memory during simulation. (Actually, this is another reason for using SIGNALS and not VARIABLES since only SIGNALS have EVENTS defined on them in VHDL.)

    Normally the EVENT to use as a trigger would be an EVENT on the enable signal (see also [14] and [15]):

```
IF enable'EVENT AND enable = '0' THEN
    .......    -- memory model
END IF;
```

    Since the enable signal is always enabled, this can no longer be the case. An EVENT on another signal must be the triggering EVENT for the memory. Let us review all other possibilities:

    - **Read_write signal**

      For this signal EVENTS take place when no operation on the RAM is needed (see consideration 9). Also, more seriously, there could be a need for a new operation on the RAM when no EVENT occurs on the read_write signal. For instance, when during some subsequent clock cycles the design is continuously reading (writing). Therefore the read_write signal cannot be used for triggering the memory.

    - **Data signal**

      Data is only offered to the RAM in case of a write operation. Using the data signal as trigger makes read operations unsimulatable. So the data signal cannot be used for triggering also.

    - **Address signal**

      Using the address signal for triggering implies a restriction: an *operation on the same position* in the register (and thus in the RAM) can only be implemented when *at least one operation on another position* is in between the two operations. This however does not present the behaviour of a real RAM. So the address signal can be used for triggering as long as the design does not violate the restriction.

    - **A combination of signals**

      A combination of several of the above signals could be used for triggering the memory. All these signals are simultaneously updated during simulation. Since there is no guarantee about the order of updating, there is no way to predict the order of the

EVENTS. Since there is no way to predict the order of the EVENTS, the simulation order of processes is undetermined. Simulations consequently become unreliable. This violates the PCALE Design Flow philosophy severely and is therefore useless.

- **Clocked RAM**

  Another possibility is the use of a clocked RAM instead of an asynchronous RAM. Here the problem arises that data is sampled at the active edge of the clock, which means that it is not possible to generate a write or read operation halfway the clock cycle in order to write data into the RAM or read data from the RAM in that clock cycle. When an operation is needed, the read_write (and data signal possibly) must be generated so that the clocked RAM copies or delivers the data in the *next* clock cycle. This implies two restrictions.

  1. Since every read operation already had to be anticipated one clock cycle, every write operation followed by a read operation must a least be separated by three clock cycles in order to perform the operations.
  2. Write operations must be delayed one clock cycle. Read operations must be completed one extra clock cycle earlier to a total of two clock cycles.

  As both restrictions are unacceptable, the RAM cannot be modelled as a clocked RAM.

Apparently, the only useful solution is the use of the *EVENTS on the address signal as the triggering EVENTS* for the memory model during simulation. This means that the design must comply with the restriction for separating operations on the same position.

However, this restriction can be relaxed a little. Since data is read before the clock cycle in which the data is needed, this data has to be retained during the active edge of the clock. This is done by sampling the data in a sampling register. This register has a size equal to the wordlength of the RAM. This sampling register allows for two read operations on the same address after each other. No EVENT occurs for the address signal, but the needed value is still present in the sampling register.

From the above considerations, the following restrictions are deducted:

- The number of operations per clock cycle is limited to one.
- Read operations are anticipated one clock cycle at least. Write operations are not anticipated.
- A read (write) operation after a write (read) operation or vice versa or two write operations after each other on the same address, are separated by at least one operation on another address. Two read operations after each other on the same address are allowed.

These restrictions are further referred to as the *replacement restrictions*. Only if a design complies with these replacement restrictions, can the design be converted while preserving design functionality.

The considerations of this chapter form the basis for the VHDL models of the memory and the addressgenerator. The VHDL model of the memory is listed in appendix E. An *example* of the VHDL model of the addressgenerator is listed in appendix F since the address-conversion in the addressgenerator is different for every design and RAM.

4.3. Adjustments for simulation

When the replacement tool performs the register replacement, the HL VHDL description has to be changed in several ways. The following adjustments are needed to obtain the descriptions that can be simulated to verify functional correctness after replacement:

- **Insertion of package declaration and package creation**

  Type conversion functions from integer to std_(u)logic and vice versa are needed for the replacement. These conversion functions are defined in a package. Some symbolic constants also are defined in this package; the constants are used in the inserted VHDL code because they enhance readability. This package has to be made "visible" to the entity in which the register has been replaced through a package declaration. The package itself has to be created with the correct contents, for instance the correct constant value for the response time of the RAM has to be defined in the package.

- **Insertion of component declarations and configurations**

  The VHDL descriptions of the RAM and of the addressgenerator are defined in a separate file. Since these descriptions are referred to as components in the new HL description, both declaration and configuration of these components have to be inserted at the correct position in the altered entity.

- **Insertion of signals for communication with RAM**

  The signals that stem from the design and that control the RAM (which is added as a component in the design), must be declared inside the entity.

- **Insertion of component instantiations**

  The declarations and configurations of the RAM and the addressgenerator are defined in the entity header. The actual instantiations of these components are also added as VHDL statements to the entity. The instantiations are concurrent statements and are added right after the beginning of the entity body.

- **Adjustment of register declaration**

  The original register including the declaration is removed from the entity. A new register is needed due to the replacement. This new register samples the RAM output. So the register declaration of the large register is replaced by another register declaration for the new and much smaller register.

- **Configuration of the addressgenerator**

  Perhaps the RAM is larger than the register. This means that not all bits of the address signal are used. During configuration of the addressgenerator the unused address bits are assigned a value of '0'. Furthermore in case of a one-dimensional register the index variable has to be divided by the wordlength to obtain the correct *word* address for the RAM. Actually this division consists of leaving the least significant register address bits unused after conversion to std_(u)logic: a total of ^{2}log(wordlength) bits is not used.

- **Insertion of default value for read_write signal**

  The read_write signal is default set for reading since the RAM is always enabled. The code that assigns this default value is inserted after the active edge statement.

- **Insertion of sampling statement**

  The code for assigning the output of the RAM to the sampling register is also inserted. This sampling statement is inserted after the code for the default value of the read_write signal.

The above stated alterations only need to be performed once. But there are other VHDL statements in which the register occurs and they can occur multiple times in the entity body. These VHDL statements are register assignment, register access, procedure calls and function calls. They are the most important VHDL statements to replace because they describe the functionality of the design. The next sections discuss these VHDL statements and their equivalents for the RAM.

4.3.1. Register assignment

When the replacement tool encounters VHDL code for a register assignment (when a part of the register is assigned a value), then this code has to be replaced by equivalent code that stores the data in the RAM. First of all it is important to know what the code for a register assignment looks like for one-dimensional and for two-dimensional registers. Only then it is possible to recognize the essential parts of such an assignment after which the equivalent code can be constructed.

An assignment to a one-dimensional register is a VHDL statement of the form:

```
repl_reg(index+wordlength-1 DOWNTO index) := some_value;
```

For a two-dimensional register an assignment is in a slightly different form:

```
repl_reg(index) := some_value;
```

The essential parts of such assignments are:

- **The name of the register**

  The fact that the name of the register (repl_reg) occurs in a statement and the fact that the name of the register occurs *before* the variable assignment symbol (:=), tells the replacement tool that this statement is a register assignment. Thus this statement has to be replaced by equivalent code for data storage in the RAM. In this way the name of the register to replace is only used for recognition of the register assignment.

- **What part of the register is being assigned**

  For the second essential part of a one-dimensional register assignment, what part of the register is being assigned, a choice has to be made whether to use the upper index (index+wordlength-1) or the lower index (index), since only one of them can be used as an address for the RAM. The most logical choice is the lower index since this implies a more direct address translation in the addressgenerator: the index value of 0 corresponds with address 0 of the RAM.

  This index has to be sent to the addressgenerator for translation from linear indexing to indexing on word basis. Therefore the following code is inserted (assuming that

address is the name of the signal that is used as address input signal for the addressgenerator):

```
address <= index;
```

For a two-dimensional register no choice has to be made of course. The index is sent to the addressgenerator with the same VHDL statement as for a one-dimensional register.

- **The value that is assigned**

  The last essential part of a register assignment is the value (some_value) that is assigned. This value must be sent to the RAM as data to store. Assuming that the name of the signal that is used to this end, is data_ram_in, the following code is inserted:

```
data_ram_in <= some_value;
```

Besides the above two VHDL statements, another statement has to be inserted. Since a write operation has to be performed, the read_write signal must be set accordingly:

```
r_w <= WRITE;
```

So the equivalent RAM code for a register assignment is nothing more than three statements: one statement for the data itself, one statement for the address to store the data and one statement to signal the RAM to perform a write operation.

4.3.2. Register access

Similar to register assignment is the case of register access: when the replacement tool encounters VHDL code for a register access (when a part of the register is accessed), then this code has to be replaced by equivalent code that retrieves the data from the RAM.

The VHDL code for accessing a one-dimensional register can be in one of the two following forms:

```
some_signal <= repl_reg(index+wordlength-1 DOWNTO index);
```

or:

```
some_variable := repl_reg(index+wordlength-1 DOWNTO index);
```

In case of a two-dimensional register the equivalent statements are:

```
some_signal <= repl_reg(index);
```

or:

```
some_variable := repl_reg(index);
```

Note that the first statements of the above pairs are NOT concurrent signal assignments. As stated in section 4.2, the register is assumed to be a VARIABLE. Hence they are to be interpreted as a sequential statements and not as concurrent statements.

For a register access there are only two essential parts that determine how the replacement is to be performed:

- **The name of the register**

  It does not matter whether the register value is assigned to a signal or a variable: when the name of the register (repl_reg) is encountered *after* a signal assignment symbol (<=) or *after* a variable assignment symbol (:=), the tool decides that a register access has been found.

- **What part of the register is being accessed**

  Register access means that a read operation is performed on the RAM. As already has been mentioned in section 4.1, read operations have to be anticipated at least one clock cycle. This is done by updating the index variable in the clock cycle before the data is needed. This way the RAM starts retrieving the correct data before the clock cycle in which the data is needed; at the start of a new clock cycle, the data is available on the output of the RAM and is stored in the sampling register. Assuming that the original VHDL code complies with those rules, only one the statement is needed to replace the register access. This statement only concerns retrieving from the sampling register. The sampling itself is performed every clock cycle.

  Therefore the following code replaces the register access (assuming that the sampling register is named sample_reg):

```
data_read <= sample_reg;
```

VHDL code for the read_write signal to indicate the RAM to perform a read operation, does not have to be inserted since the default value for the read_write signal is set to reading (see sections 4.1 and 4.3).

So the equivalent RAM code for a register access is nothing more than one statement: the statement for assigning the read data that is stored in a sampling register.

4.3.3. Procedure calls

It is of course possible that the register to replace does not occur in a direct assignment or access, but as parameter in a procedure call. Replacing the original register code with equivalent RAM code becomes somewhat more complex in this case.

The following aspects have to be considered in case of a procedure call, namely:

- The *total* register can be parameter in the procedure call. However, an operation on the RAM involves one word in the RAM which corresponds with *part* of the register. So replacement implies figuring out a couple of things:

  1. It has to be determined what part of the register is involved in the procedure call. It might even be impossible to determine this, for instance when this part is determined on basis of incoming data.
  2. Possibly that part is larger than the wordlength of any available RAM. This means that the restriction of one operation per clock cycle is violated. Hence, replacement is useless since it leads to erroneous results.

3. The design description can contain several procedure calls to the same procedure. In that case it has to be determined whether every procedure call involves the same part of the register and if every procedure call has the register to replace as parameter. If not every procedure call contains the register to replace and the same part of that register, then separate procedures have to be created for every individual case.

- When the parameter mode of the register is IN, the value of the register that samples the output of the RAM every clock cycle is needed inside the procedure body. So this register must become an IN parameter.
- When the parameter mode of the register is OUT, the data_ram_in signal and the r_w signal must become SIGNAL parameters of the procedure. An alternative is that the values for these signals are assigned to variables in the procedure body and that the values of these variables are assigned to the signals *after* the procedure call.
- When the parameter mode of the register is INOUT, this means a combination of both IN and OUT.
- The index variable is needed as index for the register and must be assigned a new value for the next clock period in case a register access occurs in the next clock period. Hence, the index variable must become an INOUT parameter in case a register access it to take place in the next clock period.

From the above list can be concluded that conversion of procedure calls and corresponding procedures is not impossible but that it is very difficult and laborious, while removing the procedure call and inserting the procedure body is much easier.

Therefore the corresponding procedure body is inserted in case a procedure call with the register to replace as parameter is encountered. However, a procedure may have local declarations, such as VARIABLE declarations. These declarations cannot be copied to the position of the procedure call, since declarations cannot be done in the statement part of a process. They must be done inside the declarative part of the calling process. This problem is solved by inserting the declaration part of the inserted procedure in the declarative part of the calling process. And finally, of course, the register replacement has to be redone for inserted procedure bodies.

4.3.4. Function calls

The same observations that were made for procedure calls can be made for function calls; however, since in VHDL all parameters of a function are of mode IN, it is impossible to change the parameter list of a function in the sense that the parameter list is adjusted to support the replacement. Thus function calls with the register to replace as parameter are not allowed. In case the replacement tool detects a function call with the register to replace as parameter, the replacement is not performed and an error message is generated.

4.4. Adjustments for synthesis

In section 4.3 all the adjustments that have to be made in the original HL description to obtain simulatable files have been discussed. However, before synthesis can be attempted with a synthesis tool, the RAM has to be removed from the description since this part of the description is not synthesized: an existing IC is used for the RAM component in the entity. The tool actions needed to remove the RAM from the description and to obtain a synthesizable description of the rest of the design are reviewed in this section.

The tool actions needed to create the obtain the description to be synthesized by a synthesis tool are:

- **Commenting out RAM component declaration, configuration and instantiation**

  Of course, the RAM component has to be removed completely from the entity. This means that component declaration, configuration and instantiation have to be removed. To allow for the designer to be able to trace the actions of the replacement tool, the removal is done by making these VHDL lines comment.

- **Removal of signal declarations of signals for communication with RAM**

  The signals that control the RAM are not local signals of the entity anymore since the RAM component is removed; consequently their declarations have to be removed.

- **Addition of signals to port interface for communication with RAM**

  The RAM is added as a component *inside* the original design ENTITY. This means that the port interface of the ENTITY remains unaltered. As a consequence the testbench that is used for simulation can be used both before and after replacement.

  If the RAM is added as a component *outside* the original design ENTITY, then the port interface of the ENTITY has to be altered: all signals for communication with the RAM have to become port SIGNALS. Since then the testbench has to be altered too, comparison of simulation results before and after replacement becomes more difficult and laborious. Therefore the first approach of adding the RAM *inside* the original design is taken.

  But before synthesis the RAM is removed as component from the design. The RAM control signals are also removed from the entity. These signals must become interface parameters of the entity. Hence the entity header must be extended with these signals.

4.5. Conclusions

A large register in a design requires a large memory capacity to be available in flexible hardware when a bread board is developed for the design. Flexible hardware elements in contrast have very little memory capacity. Therefore synthesis tools have to use a lot of flexible hardware elements when mapping a design containing large registers to flexible hardware. This means that a design has to be partitioned among multiple elements.

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 preferable 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 this the synthesis tools crash 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 possibilities to change a design containing a large register into design containing a RAM have been investigated. As it turns out, the replacement of a register by a RAM while preserving the functionality of the original design can be done if the register complies with certain restrictions.

These restrictions, called the *replacement restrictions*, are:

- The number of operations per clock cycle is limited to one.
- Read operations are anticipated one clock cycle at least. Write operations are not anticipated.
- A read (write) operation after a write (read) operation or vice versa or two write operations after each other on the same address are separated by at least one operation on another address. Two read operations after each other on the same address are allowed.

The replacement changes the emulation flow. After the replacement and functional verification of the result of the replacement, the RAM part of the design is removed to enable implementation of the rest of the design in flexible hardware. At the end of the emulation flow a bread board is built from the flexible hardware and an existing RAM IC. Emulation of this bread board has to establish functional correctness of the final result.

5. Templates

Register replacement by a RAM can only be performed under certain restrictions, as has been discussed in the previous chapter. In general, register specifications do not comply with the restrictions since they are quite severe and differ a lot from the normal, instinctive use of registers. Also VHDL code can be so complex that a register might comply with the restrictions, but writing a tool capable of checking the restrictions is much too complex in the general case (for instance in case of concurrent statements). However, it is desirable to know in advance whether the restrictions are met; that is, to know how to describe a register in a way that it is guaranteed that the replacement can be done. To this end templates are written so that, when these templates are used in a correct manner, the replacement under preservation of design functionality is then guaranteed. The replacement tool has to check the design for compliance with these templates.

In chapter 4 the differences in behaviour between registers and RAMs have been investigated. The evaluation of those differences lead to the restrictions under which the replacement can be performed under preservation of design functionality. Several templates are needed to guarantee compliance with the replacement restrictions. In short the templates and their relation to the replacement restrictions are:

1. **Templates for register declaration**

   The templates that involve the declaration of the register are needed since there are several ways to describe a register in VHDL and only two of them are supported by the replacement tool: one template for one-dimensional registers and another template for two-dimensional registers.

   In connection with the register, an index variable is needed. The declaration of this index variable is also described in the templates. Furthermore, the replacement tool needs two constants that indicate the length of the register and the size of the slices that are indexed. Both constants enhance readability of inserted code, and they are described in the templates, too.

2. **Templates for register assignment and access**

   These templates describe the correct manner of register assignment and access. On top of that, the templates guarantee that the index value of the register is known one clock period (or more) in advance in case of register access.

3. **Templates for IF-statements and for CASE-statements**

   These templates ensure one register operation maximally during one clock cycle. Verification of this restriction is very complex in general. Such a verification involves dataflow analysis, and it is doubtful whether dataflow analysis is possible for VHDL concurrency. The best that can be done is stating templates for which verification is realizable. Correct use of these templates guarantee that the restriction is met, but the templates do not cover every possible way to describe a register correctly.

4. **Other guidelines**

   The restriction that two operations (except two read operations) on the same address are separated by an operation on another address, cannot in general be captured in tem-

plates. Verification of the restriction is comparable with the verification of at most one register operation during one clock cycle. Therefore some informal guidelines are given to help the designer in describing a large register in a design. The designer takes responsibility for compliance with the replacement restrictions. The templates are intended as an aid to the designer.

The templates and the guidelines are discussed in more detail in the next sections.

5.1. Templates for register declaration

Since there are several ways to describe a register in VHDL and only two of them are supported by the replacement tool, the declaration of the register to replace has to be prescribed by templates. The replacement tool supports one-dimensional registers that comply with the following template:

Template for declaration of one-dimensional register

```
CONSTANT wordlength      : INTEGER := any_positive_constant_value;
CONSTANT registerlength  : INTEGER := any_positive_constant_value;

VARIABLE repl_reg        : std_ulogic_vector(registerlength-1 DOWNTO 0);
VARIABLE index           : INTEGER RANGE 0 TO registerlength-1;
```

FIGURE 18. Template for declaration of one-dimensional register

Two-dimensional registers that are supported by the replacement tool are prescribed by the following template:

Template for declaration of two-dimensional register

```
CONSTANT wordlength      : INTEGER := any_positive_constant_value;
CONSTANT registerlength  : INTEGER := any_positive_constant_value;

TYPE reg_type IS ARRAY(0 TO registerlength-1) OF std_ulogic_vector(wordlength-1 DOWNTO 0);
VARIABLE repl_reg        : reg_type;
VARIABLE index           : INTEGER RANGE 0 TO registerlength-1;
```

FIGURE 19. Template for declaration of two-dimensional register

Replacement of registers of type SIGNAL is not supported. SIGNALS in VHDL offer the system designer usage of the register in multiple entities. Replacement of the register over multiple entities is not supported, therefore the restriction on registers of type VARIABLE. Also, entities describe concurrent behaviour. Verification of concurrent behaviour for compliance with the replacement restrictions is very complex and perhaps even impossible. Furthermore, registers are usually described in VHDL by means of VARIABLES and not by SIGNALS, since registers by nature possess full random access and assignment that only can be modelled in VHDL by VARIABLES.

The variable that is used as index for the register is (trivially) an integer with values within the total range of the register. This variable is denoted in the templates.

In the register and index variable declarations the constant value indicating the register-length is needed; also the size of the part of the register that is indexed at a time is needed as a constant. Therefore these two constants must be stated before the declarations in the templates. These constants are also used in some VHDL code that is inserted during replacement, for instance in the VHDL models of the addressgenerator and the memory.

5.2. Templates for register assignment

To make sure that every register assignment that is performed on the register complies with the replacement restrictions, the following template must be used whenever a one-dimensional register is being assigned:

Template for one-dimensional register assignment

```
index := new_index_value;
repl_reg(index+wordlength-1 DOWNTO index) := new_register_value;
-- do not state a new value for the index variable here
```

FIGURE 20. Template for one-dimensional register assignment

Key elements of this template are that the size of the part of the register that is indexed is equal to wordlength and that the indexing variable is updated immediately *before* a register assignment. The latter is necessary to make sure that the EVENT on the address signal that triggers the memory during simulation coincides with the new register value.

For two-dimensional registers the equivalent template is:

Template for two-dimensional register assignment

```
index := new_index_value;
repl_reg(index) := new_register_value;
-- do not state a new value for the index variable here
```

FIGURE 21. Template for two-dimensional register assignment

The size of the part of the register that is indexed is equal to wordlength due to the declaration of the two-dimensional register. Hence the only key element of this template is that the indexing variable is updated immediately *before* a register assignment similar to the template for one-dimensional register assignment.

5.3. Templates for register access

Every register access that is performed on the register has to comply with the restrictions. Therefore the following template must be used for one-dimensional register access:

Template for one-dimensional register access

```
-- do not state a new value for the index variable here
... <= repl_reg(index+wordlength-1 DOWNTO index);
-- state a new index value with the next statement ONLY if the next register operation is a register access
index := new_index_value;
```

FIGURE 22. Template for one-dimensional register access

Key elements of this template are that the size of the part of the register that is indexed is equal to wordlength and that the indexing variable is updated immediately *after* a register access if the next operation on the register is also a register access.

For a two-dimensional register the equivalent template is:

Template for two-dimensional register access

```
-- do not state a new value for the index variable here
... <= repl_reg(index);
-- state a new index value with the next statement ONLY if the next register operation is a register access
index := new_index_value;
```

FIGURE 23. Template for two-dimensional register access

The size of the part of the register that is indexed is equal to wordlength due to the declaration of the two-dimensional register. That leaves only one key element for this template namely that the indexing variable is updated immediately *after* a register access similar to the template for one-dimensional register access.

In case there is not a register access in a clock cycle while the next clock cycle contains a register access, the above templates cannot be used. In fact the only statement that is needed in this case is the updating of the indexing variable. The template for this situation is shown in figure 24. It applies to both one-dimensional and two-dimensional registers.

Template to use when the next clock cycle contains a register access

```
index := new_index_value;          -- set index variable for register access in next clock cycle
```

FIGURE 24. Template when the next clock cycle contains a register access

The indexing is not part of template checking. This gives the designer a maximum of freedom for updating the index variable. Any violation in providing the index at least one clock period before register access is noticed during simulation of the converted design.

5.4. Templates for IF-statements

The previous sections involved templates for register declaration, access and assignment. Basically these templates can only guarantee that the register is of the correct type and that the read operations on the RAM can be anticipated one clock cycle. They do not guarantee that at the most one operation is performed per clock cycle.

The simplest way to guarantee that this restriction is met, is limiting the number of register operations to one. However, writing to the register without reading or reading from the register when nothing has been written, is useless: the number of register operations cannot be limited to one. Still it is desirable to be able to guarantee that the restriction is met.

As already stated, verification of the restriction in the general case is too complex. But there are some VHDL statements for which mutual exclusiveness is guaranteed. The statements for which mutual exclusiveness is guaranteed are IF-statements and CASE-statements. This section involves the templates for IF-statements; in the next section the templates for CASE-statements are reviewed.

The register can occur in the conditions of the (ELS)IF-clauses of an IF-statement. Also, the register can occur in any THEN-clause. If the register occurs in a condition and in a THEN-clause, then two operations per clock cycle are indicated: one for the condition and one for the THEN-clause. So the templates must ensure that this is not the case.

Two templates can be given for IF-statements. The first is based on the occurrence of the register in at least one THEN-clause. This implies that the conditions of the IF-statement do not contain the register. Of course the register may not occur more than once inside one single THEN-clause. This first template is stated in figure 25.

IF-template 1: conditions do not contain register

```
IF      condition_1 THEN sequential_statements      -- at most one register operation
ELSIF   condition_2 THEN sequential_statements      -- at most one register operation
...
ELSIF   condition_n THEN sequential_statements      -- at most one register operation
ELSE    sequential_statements                       -- at most one register operation
END IF;
-- index may (optionally) be stated here for resource sharing reasons
```

FIGURE 25. Template 1 for IF-statements

If the register is needed in at least one condition, then another template has to be used. The template for this situation is stated in figure 26. Occurrence of the register in a condition implies register access; register assignment is impossible. Since register replacement involves several statements that cannot be inserted in a condition, the register access has to be performed before the IF-statement: a register value is assigned to a temporary variable. At that position the replacement can be performed. The updating of the index variable when the next register operation is a register access is denoted in the template, too. This

updating is done in the THEN-clauses or after the IF-statement for resource sharing reasons.

IF-template 2: register value is needed in condition(s)

```
VARIABLE tmp_var : std_ulogic_vector(wordlength-1 DOWNTO 0);
...
tmp_var := repl_reg(index+wordlength-1 DOWNTO index);
IF      condition_1_with_tmp_var      THEN  sequential_statements   -- no use of repl_reg;
                                                                     -- tmp_var can be used
                                           index := new_value        -- if next operation is access
ELSIF   condition_2a_without_tmp_var THEN  sequential_statements   -- no use of repl_reg;
                                                                     -- tmp_var can be used
                                           index := new_value        -- if next operation is access
ELSIF   condition_2b_with_tmp_var    THEN  sequential_statements   -- no use of repl_reg;
                                                                     -- tmp_var can be used
                                           index := new_value        -- if next operation is access
...
ELSIF   condition_n_without_tmp_var  THEN  sequential_statements   -- no use of repl_reg;
                                                                     -- tmp_var can be used
                                           index := new_value        -- if next operation is access
ELSIF   condition_n_with_tmp_var     THEN  sequential_statements   -- no use of repl_reg;
                                                                     -- tmp_var can be used
                                           index := new_value        -- if next operation is access
ELSE    sequential_statements              -- no use of repl_reg;
                                           -- tmp_var can be used
        index := new_value                 -- if next operation is access
END IF;
-- index may (optionally) be stated here for resource sharing reasons
```

FIGURE 26. Template 2 for IF-statements

The fact that the temporary variable may or may not occur in any condition is symbolically denoted in the template. Also any THEN-clause can contain the temporary variable, even multiple times. This is denoted as comment in the template. Of course none of the conditions and THEN-clauses contain the register.

There is a restriction for the usage of the IF-templates discussed in this section and the CASE-templates stated in the next section. Unnested the total of all templates cannot be more than one. Otherwise the replacement tool cannot verify the replacement restrictions by means of template checking. There is no restriction on the nested usage of the templates. If the templates are violated, the replacement tool asks the designer if the replacement still has to be performed. In that case, the designer takes responsibility for the replacement and for the fact that the replacement restrictions have to be met. It is then NOT guaranteed that the replacement results in preservation of design functionality.

5.5. Templates for CASE-statements

The templates for CASE-statements in this section are completely analogue to the templates for IF-statements of the previous section. The first CASE-template, shown in figure 27, is when the register statements occur in the WHEN-clauses and not in the expression.

CASE-template 1: expression does not contain register

```
CASE expression IS
    WHEN expression_values_1  => sequential_statements      -- at most one register operation
    WHEN expression_values_2  => sequential_statements      -- at most one register operation
    ...
    WHEN expression_values_n  => sequential_statements      -- at most one register operation
    WHEN OTHERS               => sequential_statements      -- at most one register operation
END CASE;
-- index may (optionally) be stated here for resource sharing reasons
```

FIGURE 27. Template 1 for CASE-statements

The second CASE-template, depicted in figure 28, is when a register value is needed in the expression. Identical to the second IF-template, the register value is assigned to a temporary variable and this variable can be used in the expression and in the WHEN-clauses.

CASE-template 2: register value is needed in expression

```
VARIABLE tmp_var : std_ulogic_vector(wordlength-1 DOWNTO 0);
...
tmp_var := repl_reg(index+wordlength-1 DOWNTO index);
CASE expression_with_tmp_var IS
    WHEN expression_values_1  => sequential_statements      -- no use of repl_reg;
                                                            -- tmp_var can be used
                                 index := new_value         -- if next operation is access
    WHEN expression_values_2  => sequential_statements      -- no use of repl_reg;
                                                            -- tmp_var can be used
                                 index := new_value         -- if next operation is access
    ...
    WHEN expression_values_n  => sequential_statements      -- no use of repl_reg;
                                                            -- tmp_var can be used
                                 index := new_value         -- if next operation is access
    WHEN OTHERS               => sequential_statements      -- no use of repl_reg;
                                                            -- tmp_var can be used
                                 index := new_value         -- if next operation is access
END CASE;
-- index may (optionally) be stated here for resource sharing reasons
```

FIGURE 28. Template 2 for CASE-statements

5.6. Guidelines for register replacement

The templates that are defined in the previous sections cannot be used without some precaution. Furthermore, the defined templates involve only two of the three replacement restrictions. The third replacement restriction that two operations (except two read operations) on the same address have to be separated by another operation on another address cannot in general be captured in a VHDL template. The verification of the restriction involves dataflow analysis which cannot be captured in a VHDL template. On top of that, dataflow analysis is very complex in VHDL, if it is possible at all.

Therefore some guidelines are stated in this section. These guidelines help the designer in writing a design description with a large register that can be replaced. The guidelines, which involve all three replacement restrictions, are:

1. The templates for the declaration of register, for register assignment and for register access must ALWAYS be used.
2. It is important to realise that the RAM, which is to replace the register, can only perform one read or one write operation per clock period. This means that the register also can have one operation per clock period at the most. So the designer has to make sure that the design satisfies this restriction. The templates for IF-statements and for CASE-statements can help with this restriction.
3. For simulation purposes, an EVENT is needed on the address signal that is sent to the RAM. So two operations on the same part of the register are to be separated by another operation on another part of the register. Only two read operations on the same part of the register do not need to be separated by another operation on another part of the register.
4. Since read operations on the RAM have to be performed one clock cycle (or more) before the data is needed, it is expected that the index variable is set to the correct value before the clock cycle in which the actual register operation takes place. Notice that in case of a one-dimensional register the value of the index variable is assumed to be the lower index of the address; the upper index is determined by adding the wordlength to the lower index (see the templates for register access in section 5.3).

6. Testcase

Of course, the replacement tool has to be tested to check its functionality in daily life practice. The replacement tool is to be used for the first time 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 replacement tool.

In this chapter that testcase is discussed. First the Demultiplexer/Descrambler as part of Digital TV Receivers is reviewed, then the Demultiplexer/Descrambler is looked into in some detail. The next chapter is involved with the tests themselves, including the test that has been performed with the Demultiplexer/Descrambler. The interested reader is referred to [5] for a complete description of the Demultiplexer/Descrambler.

6.1. Digital TV Receiver

Nowadays 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 [3] and [4].

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-Solomon 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 can 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 29.

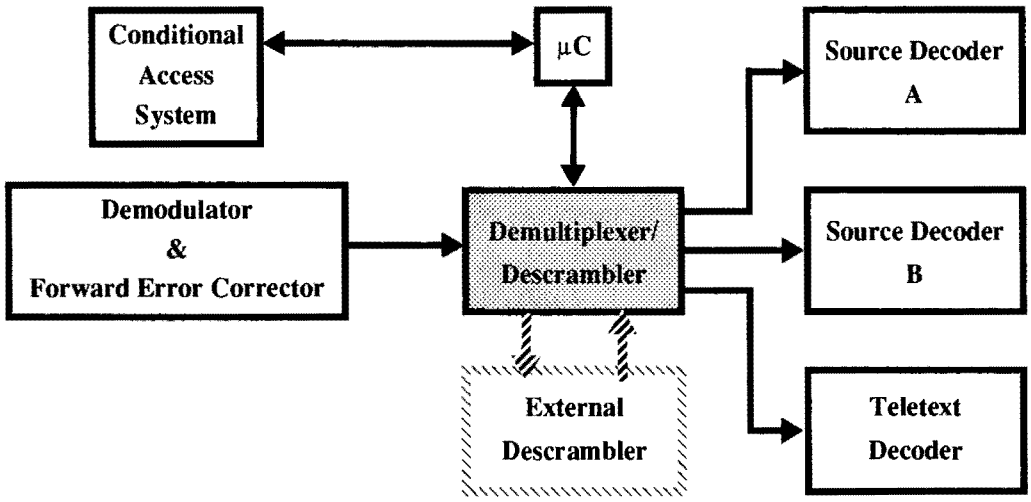


FIGURE 29. Example of a Demultiplexer/Descrambler system configuration

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. In the next section the Demultiplexer/ Descrambler is looked into in more detail.

6.2. The Demultiplexer/Descrambler

The internal structure of the MPEG-2 Demultiplexer/Descrambler is shown in the functional block diagram in figure 30. The block diagram indicates the main functional entities in the Demultiplexer/Descrambler.

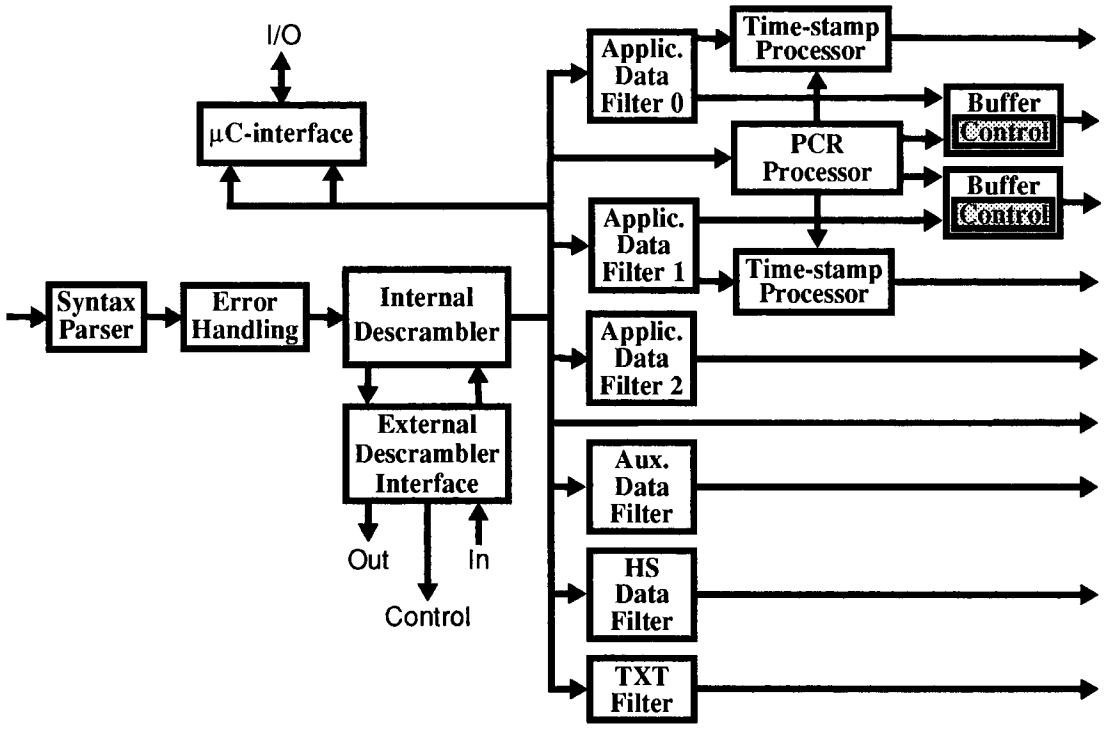


FIGURE 30. Demultiplexer/Descrambler functional block diagram

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 also obtained from the PES stream.

- **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 (PSI) and entitlement message data from the transport stream.

One or more blocks are used as testcase for DSA, for instance the PSI filter, since it contains a large register that cannot be mapped effectively to flexible hardware.

7. Testing

Evidently the tool that performs the register replacement has to be tested after its development. Testing DSA means that all the steps of the non-standard emulation flow described in figure 8 on page 21 are taken, with a suitable testcase as input for the first step. The testcase that is used to this end is the PSI filter in the micro-controller interface of the Demultiplexer/Descrambler. Correctness of the functionality of the tool is shown by comparing the simulation results before and after the register replacement and the emulation results; these results must be identical. The testcase and its relevance have been explained in chapter 6 and the test results are described in section 7.2.

However, before putting a lot of effort into the development of DSA, the principle of register replacement has to be tested first. In principle the parser of the Demultiplexer/ Descrambler could be used for such a test. But when the time had come to test the principle of register replacement, there were a number of problems that inhibited the use of the parser of the Demultiplexer/Descrambler as testcase. The problems were:

1. The parser had other synthesis problems than only a large unsynthesizable register. The DSA tool of L.P.M. van Lieshout (see [16]) for solving those problems, was still under development and could not be used. Rewriting the complete design was too laborious.
2. The register in the parser did not comply with the replacement restrictions. Again, rewriting the design was too laborious.
3. The parser requires a high clock frequency (about 30 MHz) while the fastest RAMs available at that time were relatively slow (response time of 25 ns). This leaves about 8 ns for design functionality. Looking at the complexity of the parser and especially at the data calculations before a write operation or after a read operation on the register, it is evident that it is impossible to create a bread board for the parser in combination with such a slow RAM.
4. The only available synthesis tool at that moment, Autologic, refused to synthesize constraints. This means that the synthesis tool could not be influenced with regard to timing.

Hence a simple testcase requiring a relative low clock frequency had to be written so that this could serve as testcase for the principle of register replacement. This simple testcase and the testresults belonging to it are described in the next section.

7.1. Testing the principle

As previously stated, there is a need for a simple testcase for testing the principle of register replacement prior to the development of the replacement tool. The testcase for this purpose is a design containing a register and a design containing a RAM. Of course the register in the design fully complies with the replacement restrictions derived in section 4.2. The design containing the RAM is completely analogue to the design containing the register, except for the fact that the register replacement as described in chapter 4 is performed.

This is done by hand, since this test takes place before development of the replacement tool. For a complete description of the testcase please refer to appendix G.

The main functionality of the design is formed by a state machine. In the design, in some states of the state machine the register is written, in other states the register is read. Very little time (about 8 ns) is available for design functionality. Also, constraints cannot be put on the design during synthesis since the synthesis tool does not synthesize designs with constraints. Therefore the most sensible thing to do is to make sure that the available time is not exceeded. This is done by performing no data calculations before a write operation or after a read operation. So this testcase cannot provide any guarantee about *timing aspects*; it can however indicate that the *functionality* of a register can be implemented in a RAM.

To test the principle of register replacement both designs are simulated in a VHDL simulator to check design functionality. The comparison of the two simulations shows that the replacement does not influence the functionality of the design: the two designs produce identical simulation results.

In addition to the simulations, the design with the RAM is adapted for synthesis as described in section 4.4. Basically this means that the RAM component instantiation is removed from the design and that the port interface of the design is expanded with the signals that are used for communication with the RAM. After this synthesis adaptation, the design is synthesized. Following synthesis, the synthesized design is mapped to EPLDs with the MAX+PLUS mapping tool of the Altera Corporation (see [12]): the design can be successfully mapped to an EPM7032LC44-3 EPLD. With this EPLD and a CY7C171 RAM from Cypress Semiconductor a bread board is built which is emulated. Comparison of this emulation with the previously conducted simulations shows that the bread board exhibits the same functionality as the design during the simulations.

Since all three tests (the two simulations and the emulation) produce exactly the same results, the principle of register replacement is established: the replacement of a register by a RAM while preserving the functionality of the original design can be done if the register complies with certain restrictions.

7.2. Testing the replacement tool

After the principle of register replacement was established, a tool was developed to automate the register replacement. The tests that have been performed to test this replacement tool are described in this section.

1. PSI filter

The PSI filter as part of the micro-controller interface was the first testcase for the replacement tool. The large, two-dimensional register in this PSI filter consists of 512 words of 16 bits. However there were no RAMs available with a 16 bit wordlength. This problem was reported by the replacement tool and it was solved by changing the wordlength of the available RAMs in the RAM library to 16. This means that several

RAMs have to be used parallel on the bread board. All corresponding pins of all RAMs except the data pins have to be connected while the data pins of the RAMs are connected to a *part* of the data pins of the design. For example in case of RAMs with a wordlength of 8 bit, one RAM is used for the lower 8 bits of the data while the other RAM is used for the upper 8 bits of the data.

After this change in the RAM library the replacement was attempted again. The replacement tool now reported that the active edge mark could not be found. Analysis of this problem showed that this error was due to concurrency. The register in the PSI filter was described as concurrent with the index variable. Since concurrency is not supported by the replacement tool, the replacement cannot be performed.

2. Other testcases

To show that the replacement tool performs a functionally correct replacement provided that the register is written compliant with the replacement restrictions, the testcase that was used for testing the principle was used again.

Besides the original testcase which contains a one-dimensional register, the testcase has also been described with a two-dimensional register. Both testcases were simulated in a VHDL simulator and showed identical functionality.

After simulation the two testcases were converted by the replacement tool. The tool reported a successful conversion in both cases. To verify that the replacement had not changed the functionality of the testcases, the testcases were simulated again in the VHDL simulator. These simulations showed that the functionality had not changed in either case due to the replacement.

Next the testcases were synthesized with the CORE synthesis tool and subsequently mapped with the MAX+PLUS mapping tool. Both designs can be successfully mapped to an EPM7032LC44-3 EPLD. Two bread boards were built each with one such EPLD and a CY7C171 RAM from Cypress Semiconductor. Then the bread boards were emulated. Comparison of the emulations with the previously conducted simulations showed that the bread boards exhibit the same functionality as the designs during the simulations.

Since all tests (the simulations and the emulations) produce exactly the same results, the functional correctness of the replacement tool is established. The replacement tool replaces one-dimensional and two-dimensional registers by a RAM while preserving the functionality of the original design (provided of course that the original design complies with the replacement restrictions).

8. Features

Besides the basic functionality of the tool, which is the replacement of a register by a RAM, some features are added to increase usability and flexibility of the tool. All these features are discussed in this chapter. The features are:

1. The tool is controlled by means of a control file, in which parameters can be set that influence the effect of the tool. Since the number of parameters can be rather large, they are passed on to the tool by means of a control file instead of by means of command line parameters. This control file is the first command line parameter.
2. Also the replacement tool must be provided with a library of RAMs. In this RAM library the data of the RAMs that can be used for the replacement is stored.

   However, storage of the necessary data of RAMs is not the only function of the RAM library. The RAM library also makes automatic selection of a RAM possible: when the designer does not specify a specific RAM to use for the replacement, the replacement tool attempts to select a RAM from the library.
3. Another feature is template checking. The tool automatically verifies the HL description on its compliance with the templates that were composed in chapter 5.
4. Furthermore the tool incorporates some error checking. In case the tool encounters VHDL constructs that make the register replacement impossible or that guarantee erroneous results, the tool issues an error message and ceases the replacement.
5. The tool also generates a transcript file that lists the actions of the tool. This transcript file contains the information.

The replacement tool is called by a command with the following syntax:

```
r2r <control file> <files file>
```

The first argument, the control file, is thoroughly described in the next section. The second argument is a so-called files file. The files file lists all files among which a design can partitioned. The order of listing these files is ascending in the sense that the top hierarchical level of a design is described in the last file. An example of a files file is listed in appendix I.

8.1. Tool control

The effect of the tool is determined by some essential data and by some extra, non-essential data. The essential data is passed to the tool by means of mandatory parameters. These mandatory parameters are discussed in section 8.1.1 in more detail.

The extra, non-essential data does not need to be passed on to the tool. If some non-essential data is passed on to the tool, it is passed on as optional parameters. These optional parameters are explained in section 8.1.2 along with their influence on the effect of the tool and their default values.

In appendix H an example of a control file is listed. All parameters, mandatory and optional, are embodied in this example.

8.1.1. Mandatory parameters

The register replacement can only be performed when some essential data is known. Since this data differs from design to design, it cannot be assigned default values. Therefore this data must be passed to the tool as parameters in the control file every time it is called. This subsection explains these mandatory parameters and their meaning. In the next subsection the other, optional tool parameters are discussed.

The mandatory parameters and their meaning are:

- CLOCK_FREQUENCY

  The clock frequency (in MHz) on which the total design must function must be specified in order for the tool to be able to check whether or not the RAM that is used for the replacement meets the speed requirements. It is also used as a criterium for the automatic selection of a RAM.

- REGISTER_TO_REPLACE

  With this parameter the designer can tell the tool the name of the register that has to be replaced; the replacement is almost completely based on finding occurrences of this name in the VHDL files.

- REGISTER_LENGTH

  This parameter specifies the length of the register indicated by the parameter REGISTER_TO_REPLACE. It must be specified in order for the tool to be able to check whether or not the RAM that is used for the replacement meets the memory requirements. It is also used as a criterium for the automatic selection of a RAM.

  It is of course possible that the tool automatically determines the length of the register by reading the range from its declaration. However, ranges in VHDL can be complex expressions that require a lot of tool complexity. Therefore the automatic determination of the length of the register is left out.

- INDEX_VARIABLE

  This parameter indicates the name of the variable that is used for indexing the register that is being replaced. By demanding that the designer uses a variable for indexing of the register instead of direct indexing it is guaranteed that the designer always indexes the same amount of data. Furthermore, through the use of a variable, it is always clear what address must be sent to the RAM, namely the value of this variable.

- INDEX_LENGTH

  This parameter specifies the length of the indexed slices of the register. It must be specified in order for the tool to be able to check whether the width of the RAM that is used for the replacement suffices. It is also used as a criterium for the automatic selection of a RAM.

For the same reason as with the parameter REGISTER_LENGTH, automatic determination of the length of the index variable is left out.

- ACTIVE_EDGE_MARK

  A VHDL statement for the default value of the read_write signal and a VHDL statement for assigning a value to the sampling register every clock cycle must be added when replacing the register. In order to ensure that these VHDL statements are executed every clock cycle, they are stated right after the VHDL code for detecting the active edge. This implies that the tool must know what VHDL code indicates the detection of the active edge of the clock. Every synthesis tool uses its own specific definition of what VHDL code indicates the detection of the active edge of the clock. Therefore the string to use for the detection of the active edge of the clock must be specified by means of this parameter.

- 2D_TYPE

  In order for the replacement tool to be able to check the declaration of a two-dimensional register, the name that is used for the type of the two-dimensional register has to be specified. The 2D_TYPE parameter is used to this end. For one-dimensional registers this parameter is left out of course.

8.1.2. Optional parameters

Besides the mandatory parameters reviewed in the previous section, some optional parameters can also be set in the control file. If these parameters are not specified, they are given a default value. These parameters are intended to increase tool flexibility.

The optional parameters, their meaning and default values are:

- RAM_NAME

  This parameter is used to specify the name of the RAM to use for replacement. The tool uses the data of this RAM for configuring the package file (see optional parameter PACKAGE_FILE) and for checking suitability of the RAM. When this parameter is not specified, the tool attempts automatic selection of a suitable RAM (see section 8.2.2).

- SIMULATION_FILES_EXTENSION

  Running the tool results in output files which can be used for simulation, so the result of the replacement can be checked as prescribed in the PCALE Design Flow (see section 2.1). These simulation output files have names equal to the original file names extended with an extension as indicated by this parameter. When this parameter is not set, a default value of ".sim.vhdl" is assumed.

- SYNTHESIS_FILES_EXTENSION

  Besides the simulation output files, the tool also produces synthesis output files. These synthesis output files are used for synthesizing the design after replacement and after the result of the replacement has been verified through simulation. These synthesis out-

put files have names equal to the original filenames extended with an extension as indicated by this parameter. Not setting this parameter results in a default value of ".syn.vhdl".

- TEMPORARY_FILES_EXTENSION

  When replacing the register by a RAM, the tool needs some temporary result files. The names of these files are equal to the original files extended with an extension as indicated by this parameter. The default value for this parameter is ".r2r". The main purpose of this parameter is that the temporary files of the tool can be recognized (for instance in case of a tool crash) and that the extension can be influenced so that unwanted filenames can be avoided.

- PROCEDURE_BODY_FILES_EXTENSION

  In case of a procedure call, the procedure call has to be replaced by the procedure body. Since the procedure body can reside in the same file as the procedure call and since files cannot be used for two purposes at the same time (finding procedure calls and extracting procedure bodies), the tool makes a copy of the file containing the procedure body. This copy has a filename equal to the original files extended with an extension as indicated by this parameter. As with the parameter TEMPORARY_FILES_EXTENSION, the main purpose of this parameter is recognition of the files and avoiding unwanted filenames. The default value for this parameter is ".pbf".

- PACKAGE_FILE

  The name of the package file that is created can be stated by means of this parameter. It has a default filename of "definitions.vhdl".

- PACKAGE_FRAME_FILE

  Starting with a suitable framework, the needed package file is generated by the tool. The name of the file containing this framework can be entered through this parameter The default file name for this file is "definitions.vhdl.frame".

- ADDR_GEN_FILE

  This parameter is used to state the name of the file to create the configured addressgenerator in. The default value for this parameter is "ram.vhdl".

- ADDR_GEN_FRAME_FILE

  Similar to the framework file for the package, the name of the framework file for the addressgenerator can entered. The default name for this file is "ram.vhdl.frame".

- RAM_LIBRARY_FILE

  This parameter states the name of the file containing the RAMs that can be used for replacement. This file contains the necessary data of the RAM that is used for the replacement and this file is used for the automatic selection of a RAM. Default this parameter takes on the value of "ram.library".

- TRANSCRIPT_FILE

  The actions of the tool and possibly errors and warnings are written to a transcript file so that the behaviour of the tool can be still be viewed afterwards without rerunning the

tool. The name of this transcript file is indicated by this parameter and has as default value of "transcript.r2r".

The above explained optional parameters influence the behaviour of the replacement tool. The following optional parameters influence the VHDL code that is inserted. They can be used to alter the names of the CONSTANTS, VARIABLES and SIGNALS that are involved in the register replacement. These parameters should be set when the designer does not want to use the default names or when the default names already occur in the design before it is converted by the replacement tool.

- WORDLENGTH_CONSTANT_NAME

  The replacement tool has to know the name of the constant that indicates the length of the words involved in register and RAM operations. This constant has a default name of "wordlength" which can be overwritten by setting WORDLENGTH_CONSTANT_-NAME. This parameter should be used when the default name has already been used in the design to another end.

- NUMB_OF_ADDR_BITS_CONSTANT_NAME

  Similar to WORDLENGTH_CONSTANT_NAME, the name of the constant in the package file indicating the number of address bits of the RAM can be given a value different from the default value of "numb_of_addr_bits" by means of this parameter.

- NUMB_OF_WORDS_CONSTANT_NAME

  Also the name of the constant indicating the number of words of the RAM can be given a non-default value different from the value of "numb_of_words".

- REGISTERLENGTH_CONSTANT_NAME

  The fourth constant in the package file whose name can be influenced, is the name of the constant indicating the length of the register. It has a default value of "register-length".

- SAMPLING_REGISTER_NAME

  As with the above constants, the name of the sampling register that samples the output of the RAM every clock cycle can be explicitly stated by means of this parameter; default value for this parameter is "sample_reg".

- ADDRESS_TO_ADDR_GEN_SIGNAL_NAME

  The name of the address signal that is an input of the addressgenerator can be set by means of this parameter. The default value for this parameter is "address".

- DATA_RAM_IN_SIGNAL_NAME

  The name of the signal representing the data that must be stored in the RAM can also be set. The default signal name is "data_ram_in".

- READ_WRITE_SIGNAL_NAME

  The name of the read_write signal indicating the RAM whether to read or write can be given another name than the default of "r_w".

- DATA_RAM_OUT_SIGNAL_NAME

  The name of the signal with the data that is read from the RAM can be entered through this parameter; otherwise it takes on the value of "data_ram_out".

- RAM_ENABLE_SIGNAL_NAME

  Every RAM must be enabled before an operation can be performed. The name of the signal that is used for the enabling of the RAM can be adjusted by means of this optional parameter; its default value is "ce".

- ADDRESS_TO_RAM_SIGNAL_NAME

  This parameter states the name of the signal that represents the address of the data in the RAM, and it has a default value of "address_to_ram".

8.2. RAM library

When the replacement of a register by a RAM is to be performed, the characteristics of the RAM that is used for this replacement have to be known. Of course these characteristics could be made known to the tool in the control file. However, since specifying these characteristics in the control file is very disadvantageous, the characteristics are specified in a library of RAMs. The benefits of this approach are:

- **Maintenance**

  While every design has its own specific control file and while every designer may have its own control files, the characteristics of the RAMs are present in one file only which can be shared between many designers. This way maintenance of the characteristics is a one time change in the library. Besides not only the characteristics of RAMs can be easily updated, but new RAMs and obsolete RAMs can be added and removed very easily too.

- **Possibility of automatic selection**

  When the characteristics of the RAM to use for the replacement are specified in the control file, only the characteristics of one RAM are known to the tool, while a RAM library can contain the characteristics of many different RAMs. The presence of (the characteristics of) multiple RAMs offers the possibility of automatically selecting a RAM from the RAM library: the designer only has to specify the clock frequency of the design and the size of the register after which a suitable RAM is extracted from the library (see section 8.2.2).

- **Ease of use**

  The characteristics of every RAM only have to be inserted in the RAM library once. After that the designer only has to know the name of the RAM when he wants to force the tool to use a specific RAM; or the designer can exploit the possibility of automatic selection when the designer has no knowledge of the RAMs.

The structure of the library is discussed in section 8.2.1, while the automatic selection from the library is reviewed in more detail in section 8.2.2.

8.2.1. Structure of the library

The structure of the RAM library is simply a list of RAMs. An example of a RAM library is listed in appendix J. For every RAM in the library there are some characteristics that must or can be set. These characteristics are:

- RAM_NAME

  In order to be able to distinguish between the characteristics of each RAM, every RAM must have its own unique name, for instance the part number code that every IC has. This name must be the first characteristic of every RAM in the library: every line after the name of a RAM is assumed to state a characteristic of that RAM until the name of the next RAM in the library is encountered. All characteristics of a RAM other than RAM_NAME, can be stated in random order.

  Stating the name of a RAM in the control file forces the tool to use that RAM for the replacement; not stating a name of a RAM in the control file results in automatic selection of a suitable RAM. Any string can be used for this characteristic.

- WORD_LENGTH

  The RAM is used to store words in. The length of these words is indicated by the characteristic WORD_LENGTH. No default value is assumed for this characteristic since it differs from RAM to RAM. It is mandatory to specify this characteristic and it must be stated as an integer larger than zero.

- NUMB_OF_ADDR_BITS

  Every word in the RAM has its own specific address. This address consists of as many bits as indicated by NUMB_OF_ADDR_BITS. The total number of words in the RAM is of course $2^{\text{NUMB\_OF\_ADDR\_BITS}}$. Hence NUMB_OF_ADDR_BITS does not have to be specified as long as the parameter NUMB_OF_WORDS is set for the RAM. When both are set, they must be consistent of course. All integers larger than zero are valid values for this characteristic.

  In the package a CONSTANT is declared with the value of NUMB_OF_ADDR_BITS. This CONSTANT is symbolically used in the inserted VHDL code, for instance in the declaration of the address signal. The VHDL code for this declaration is of the following form:

  SIGNAL address_in : std_ulogic_vector(numb_of_addr_bits - 1 **DOWNTO** 0);

  Of course, the *value* of NUMB_OF_ADDR_BITS_CONSTANT_NAME (one of the optional parameters in the control file) is inserted in the above declaration instead of "numb_of_addr_bits".

- NUMB_OF_WORDS

  The number of words in the RAM is indicated by this characteristic. The total number of words in the RAM is of course equal to $2^{\text{NUMB\_OF\_ADDR\_BITS}}$. Hence the characteristic NUMB_OF_WORDS does not have to be specified as long as the above mentioned characteristic NUMB_OF_ADDR_BITS is set for the RAM. When both are set, they must be consistent of course.

This characteristic is not used in the inserted VHDL code. It is used to verify or deduct the characteristic NUMB_OF_ADDR_BITS; valid values are positive powers of 2.

- ENABLE_VALUE

  The RAM can only perform an operation when it is enabled. The characteristic called ENABLE_VALUE specifies whether enabling of the RAM is high active or low active. When this characteristic is not specified, it is assumed to be low active. Low activity is indicated by the value '0', high activity by the value '1'.

- RESPONSE_TIME

  This characteristic specifies the response time of the RAM, which is the time the RAM takes to perform one operation. It is mandatory that this characteristic is set for every RAM, since it is needed for the simulation of the HL description after the register replacement and since it differs from RAM to RAM.

  Any positive integer can be used to state the value for RESPONSE_TIME in nanoseconds. Stating 0 nanoseconds as the value for the response time of a RAM is allowed. However, simulations with this value are incorrect since in reality RAMs always take a certain amount of time for an operation. The reason for allowing this value is that the designer may want to use it for comparison with register simulations. Register operations do not take time in simulations.

- READ_VALUE

  This characteristic indicates what value has to be assigned to the read_write signal of the RAM in order for the RAM to perform a read operation. READ_VALUE can take on the value of '0' or '1'.

  Note: it must be opposite to the value of the characteristic WRITE_VALUE; when WRITE_VALUE is specified, READ_VALUE does not need to be specified; when both characteristics are not specified, READ_VALUE assumes the default value of '1'.

- WRITE_VALUE

  This characteristic indicates what value has to be assigned to the read_write signal of the RAM in order for the RAM to perform a write operation. Just like the characteristic READ_VALUE, WRITE_VALUE can take on the values '0' or '1'.

  Note: it must be opposite to the value of the characteristic READ_VALUE; when READ_VALUE is specified, WRITE_VALUE does not need to be specified; when both characteristics are not specified, WRITE_VALUE assumes the default value of '0'.

- STATUS

  This "characteristic" can take on three values: AVAILABLE, ON ORDER or NOT AVAILABLE. When it is not specified for a RAM, it takes on the default value of AVAILABLE which simply means that the RAM is available. However, a RAM with a STATUS of ON ORDER causes a warning to appear for the designer that the RAM is currently not available but on order; a STATUS of NOT AVAILABLE causes a warning that the RAM is not available. The STATUS characteristic is also used for the automatic selection of a RAM (see section 8.2.2).

8.2.2. Automatic selection from the library

When the designer does not specify a specific RAM to use for the replacement in the control file, DSA attempts automatic selection of a RAM in the RAM library that best matches the requirements imposed on an adequate RAM. To this end, DSA starts with the creation of a list of all the RAMs in the RAM library. Then the automatic selection starts. The five consecutive steps in the process of automatic selection are:

1. All inadequate RAMs are removed from the list. Adequacy of a RAM is based on three relations. The first relation verifies whether a RAM is fast enough in relationship to the clock frequency:

$$response\_time < \frac{1000}{clock\_frequency}$$

   The second relation is to determine whether the wordlength of the RAM equals the length of the index variable since the tool is (partially) based on this equality:

$$word\_length = index\_length$$

   The third relation that is used to determine adequacy involves the memory capacity. The memory capacity of the RAM must at least be equal to the memory capacity that is needed by the register. So for a one-dimensional register the following relation must hold:

$$word\_length \times number\_of\_words \geq register\_length$$

   In case of a two-dimensional register another relation is used:

$$number\_of\_words \geq register\_length$$

   Only if a RAM suffices the first two relations and the appropriate third relation, the RAM is considered adequate.

   When this step leaves no RAMs in the list, this means that the replacement cannot be performed since there is no adequate RAM. When only one RAM remains after this selection step, this RAM is automatically selected since apparently this is the only RAM that suffices the requirements. When there is more than one RAM that meet the requirements, the next selection step is taken.

2. The second step in the selection process is based on the response time of the RAMs. Since a large amount of the available time within a clock period is consumed by the RAM, and since it is desirable that as much time as possible is available for design functionality (in order to get the highest chance of success for synthesis), the fastest RAM is selected.

3. When there are more than one adequate and equally fast RAMs available in the RAM library, the next selection step is taken. The RAMs with the smallest memory capacity are selected since in general larger RAMs cost more area on the final bread board (which must be kept as small as possible of course).

4. The last criterium on which a RAM is automatically selected, is the STATUS of the RAM. The selection is made in the following fashion: available RAMs are preferred over RAMs that are on order which in turn are preferred over not available RAMs. So when either all the RAMs have the same status or when the status of every RAM is not set, this step results in no further selection.
5. When after all the previous steps in the selection process, DSA still cannot decide what RAM to use for the replacement, the designer is asked to enter what RAM to use for the register replacement. The RAM that is then indicated by the designer must meet the requirements. Otherwise the designer is asked to re-enter a RAM until an adequate RAM has been entered.

   Possibly this adequate RAM is not the most optimal RAM. In that case the designer is given the opportunity to change his mind. Having changed his mind or not, the register replacement then proceeds.

8.3. Template checking

A design is verified on its compliance with the stated templates in the sense that a check is performed whether or not it can be guaranteed that the register to replace does not have two operations within one clock period. If this guarantee cannot be assured, then the replacement tool asks the designer whether or not to replace the register since the designer might be able to provide the guarantee. The templates are not checked for the index variable in order to give the designer complete freedom to determine the address of the next RAM operation.

8.4. Error checking

During a run of the replacement tool VHDL code is parsed, and while this takes place some checks are performed. These checks can lead to warnings when something erroneous is suspected; when the tool is sure that the register replacement cannot be performed or leads to erroneous results, this results in an error message and then the tool stops running without performing the replacement.

Warnings occur in the following situations:

- **Syntax errors in the RAM library**

  When the replacement tool encounters syntax errors in the RAM library this is reported to the designer. Syntax errors are ignored and have no effect on the replacement tool. Correct syntax for the RAM library is: <characteristic> <value>.

- **Syntax errors in the control file**

  Similar to syntax errors in the RAM library, syntax errors in the control file are reported to the designer. Again, syntax errors are ignored and have no effect. Correct syntax for the control file is: <parameter> <value>.

- **Default values for RAM parameters**

  When default values are assumed for optional RAM parameters a warning is issued on what value is assumed for which parameter.

- **Zero response time of RAM**

  When the response time of a RAM is specified as zero the designer is warned that simulation of the design after replacement is in conflict with reality. In reality, RAMs always need a certain (positive) amount of time to perform an operation.

- **STATUS parameter of RAM**

  When the STATUS parameter of a RAM indicates that the RAM is on order or not available a warning is issued, stating the value of the STATUS parameter.

Errors occur in a situation of:

- **Missing mandatory parameters in the control file**

  When mandatory parameters are missing in the control file the replacement cannot be performed. An error message is generated, indicating which parameters are missing.

- **Illegal values for parameters in the control file**

  When a parameter is stated with an illegal value the parameter and the illegal value are reported in an error message.

- **Identical file extensions and file names in the control file**

  Several file extensions and file names are optional parameters in the control file. The tool can only function correctly when these parameters do not have identical values.

- **Incorrect active edge mark**

  The tool scans the process that contains the register to replace for the occurrence of the active edge mark. When the specified active edge mark cannot be located the replacement is impossible. An error message reports this error.

- **Function call contains register**

  When the register to replace is parameter in a function call the replacement is not performed. An error message stating the function call is generated.

- **Function call in procedure call**

  Procedures bodies are inserted when a procedure call with the register to replace as parameter is encountered. This insertion is not supported when one of the parameters of the procedure call is a function call. However, this situation is detected and reported.

- **Missing mandatory characteristics in the RAM library**

  When one or more mandatory characteristics are missing for a RAM in the RAM library this situation is reported to the designer. An error message is generated, indicating which characteristics are missing.

- **Illegal values for characteristics in the RAM library**

  When a characteristic is stated with an illegal value the characteristic and the illegal value are reported in an error message.

- **Inconsistently specified RAM characteristics**

  The characteristics READ_VALUE and WRITE_VALUE and the characteristics NUMB_OF_WORDS and NUMB_OF_ADDR_BITS must be specified consistently. When they are inconsistently specified, an error message is generated stating the inconsistency.

- **Characteristic RAM_NAME is not the first specified characteristic**

  When the first specified characteristic of a RAM in the RAM library is not the characteristic RAM_NAME, then the tool reports this.

- **Empty RAM library**

  The RAM library must at least contain one RAM. The replacement tool verifies whether this is indeed the case.

- **Impossible automatic selection due to insufficient RAMs**

  During automatic selection of a RAM from the RAM library, every RAM is checked for compliance with the requirements imposed by the design. When none of the RAMs in the RAM library complies with the requirements, register replacement is impossible.

- **RAM stated in control file does not exist in RAM library**

  When the RAM specified by the designer in the control file cannot be located in the RAM library, the tool notifies the designer of this error.

- **Incorrect declarations of register to replace and index variable**

  The declarations of the register to replace and of the index variable must be according to the templates that were stated for these declarations (see section 5.1). Any violation of these templates results in an error.

- **Name of register occurs in more than one entity**

  When the name of the register to replace occurs in several entities the tool cannot decide which one of these registers has to be replaced. Therefore this situation is also recognized as error.

In general, the errors or warnings do not occur. Only when the designer makes mistakes or when the RAM library file is adapted the stated errors or warnings appear.

8.5. Transcript file

All messages generated by the replacement tool are written to a transcript file so that the designer can review the actions undertaken by the tool. The name of this file can be stated in the control file (see section 8.1). When it is not stated in the control file the default name "transcript.r2r" is assumed by the tool. The messages generated by the tool include:

- error and warning messages
- messages on the files that are opened
- messages on automatic selection of a RAM from the library

- messages on template checking
- messages on inserted VHDL code
- messages on register assignments, register accesses, procedure calls and function calls
- messages on insertion of procedure bodies and insertion of declaration lines

The messages are self-explanatory. All these messages are also shown on screen while the tool is running and the replacement takes place.

9. Conclusions and recommendations

Various conclusions can be drawn and various recommendations can be given for the aspects involved in the development of an emulation flow for designs with large memory requirements. The conclusions and recommendations are discussed in this chapter in relation to these aspects.

The conclusions are:

- **VHDL**

  Unfortunately, all VHDL constructs available for handling actions and reactions between entities (concurrent statements) are not synthesizable. Delay caused by an external device can only be accounted for in synthesis tools at the beginning or at the end of a clock cycle by means of constraints. Delay in the middle of a clock cycle (action and reaction between ports) is impossible in synthesizable VHDL and in constraints.

- **Synthesis libraries**

  VHDL is not suited as basis for synthesis library building blocks. One of the major causes for this is that the number of ways to describe the same functionality is almost infinite.

- **Register replacement**

  Under severe restrictions the register replacement can be performed. If a register is described in a design according to the restrictions, then the replacement can be performed while preserving design functionality. The restrictions are:

  1. One register operation during one clock cycle
  2. Read operations are anticipated
  3. Operations on the same address are separated by another operation on another address

- **Templates**

  The templates that have been given cover the guarantee on one register operation at a time; that is correct use of the templates guarantees that there is one register statement at the most during one clock cycle and that read operations are anticipated. The restriction on separation of operations on the same address cannot be incorporated in the templates since it requires dataflow analysis. This dataflow analysis is very complex and maybe even impossible in VHDL.

- **PCALE Design Flow**

  Emulation is a useful extension to the Existing PCALE Design Flow. The fact that the mapping results cannot be verified through simulation is an acceptable deviation of the philosophy behind the PCALE Design Flow, since the final verification through emulation is still present in the design flow. Also, between mapping and bread board building, no design actions take place so not simulating the mapping result but emulating the bread board is acceptable.

The recommendations are:

- **Synthesis libraries**

  It is desirable to have VHDL support for synthesis libraries. VHDL support makes re-evaluation of the synthesis tools worthwhile.

- **Replacement tool**

  The tool could be extended to support the replacement of registers that are declared as SIGNAL too, instead of only supporting the replacement of registers that are declared as VARIABLE. This implies that register statements can occur in multiple entities. Hence the tool would have to be extended in a way that the replacement is done not only within one entity, but over multiple entities. The fact of multiple entities has also implications with regard to the templates: the templates still hold inside one entity, but templates cannot be extended to cover the guarantee over multiple entities, since it is not feasible to exclude the concurrency that comes with multiple entities.

  The tool currently supports indexing from 0 to some positive value. The tool can be extended to support any indexing range. Since the addresses for the RAM must start at 0, the addressgenerator must be adapted because a direct translation is then no longer valid; the register addresses have to be linearly shifted in the addressgenerator over a distance of the lowest index value.

  The tool could be extended to support other RAMs too, for instance RAMs with separate read and write signals. Also RAMs with a wordlength different from the length of the index variable can be considered. In case the RAM wordlength is larger, the unused part of the words in the RAM can be stuffed with zeros. In case the RAM wordlength is smaller, several RAMs can be used parallel.

  The replacement, the replacement restrictions, the replacement tool and the templates are all based on full random access since this is the most general case. However, in some occasions a FIFO might be sufficient instead of a full random access RAM. Possibly the restrictions for replacement then become less severe. So further investigation is needed for this kind of replacement.

- **Templates**

  In principle, the replacement tool can be extended to support better template checking. At this moment the tool only takes into consideration where register statements occur and tries to guarantee the replacement restriction from that evaluation, without looking at the conditions under which the statements occur. If these conditions are also taken into consideration, then the guarantee that the restrictions are met can be given for more designs. Furthermore, when the tool also supports register of type SIGNAL, the "template" checking can be extended to check concurrency too. However, the concurrency makes this task very complex and it is not clear how this can be done. This kind of verification, which in fact is dataflow analysis, might even be impossible within VHDL.

- **Synthesis tool**

  Preferable constraints are not entered by the designer when synthesis is performed, but are automatically derived from VHDL statements. When the synthesis tool can be told to synthesize *part* of a design and to take into account the *timing* of the part of the

design that is not synthesized, then this is much more flexible than splitting the design description before synthesis.

Also it is desirable that the synthesis tool has VHDL output, thus enabling functional verification after synthesis.

- **Mapping tool**

  Preferably the mapping tool has VHDL output. Then functional verification is possible after the mapping step in the emulation flow.

- **PCALE Design Flow**

  The philosophy behind the PCALE Design Flow is based on two principles: specification and verification. The latter principle is not fully incorporated in the developed emulation flow. Therefore it is most preferable that all tools have VHDL output. Then the PCALE Design Flow philosophy can always be applied and no level ever has to be skipped with regard to verification. Currently the mapping result is simulatable, but only visual comparison with the other tool output is possible.

Appendix A. List of References

[1] Van den Hurk, J.A.A.M.
Evaluation of the PCALE VLSI Design Flow for HDTV ICs.
Report of a second phase project.
Eindhoven: Instituut Vervolgopleidingen, Technische Universiteit Eindhoven,
1992. ISBN 90-5282-189-5

[2] IEEE Std 1076-1987, IEEE Standard VHDL Language Reference Manual.
New York, The Institute of Electrical and Electronics Engineers, Inc.,
March 31, 1988.

[3] MPEG-1 Working Draft.
ISO/IEC DIS 11172
May, 1992.

[4] MPEG-2 Working Draft.
ISO/IEC JTC 1/SC 29 N 658
December, 1993.

[5] Van den Hurk, J.A.A.M.
Tentative Device Specification of the MPEG-2 Demultiplexer/Descrambler.
Internal Laboratory Report.
Eindhoven: Philips Semiconductors, Dept. PCALE,
November 8, 1993. Draft version 2.0

[6] Autologic VHDL Synthesis Guide.
Wilsonville, Oregon: Mentor Graphics™ Corporation,
1993. Software Version 8.2

[7] Autologic Library Development Manual.
Wilsonville, Oregon: Mentor Graphics™ Corporation,
1993. Software Version 8.2

[8] CORE User Manual.
Berkeley, CA: Exemplar Logic™,
1993. Software Version 1.21

[9] LBuild User Manual.
Berkeley, CA: Exemplar Logic™,
1993. Software Version 1.2

[10] LGen User Manual.
Berkeley, CA: Exemplar Logic™,
1993. Software Version 1.21

[11] Van der Horst, E. and P. Van der Haar and R. Krikhaar.
VHDLSyn™ v1.0 User Manual.
Hilversum (The Netherlands), Philips Electronic Design & Tools,
April 5, 1993. Software Version 1.0, EDTH-SYN-UM-022

[12] User Guide MAX+PLUS II
San Jose, CA: Altera™ Corporation,
1992. Software Version 3.0

[13] Dutt, N. D.
Generic Component Library Characterization For High Level Synthesis.
In: Proceedings, Fourth CSI/IEEE International Symposium on VLSI Design, 1991.
(Cat. No. 91TH0340-0), p. 5-10

[14] Bink, J. M.
Data Structures and VLSI.
Faculty of Electrical Engineering, Section of Digital Information Systems,
Eindhoven University Of Technology, August 1991.
Master's Thesis no. 5864

[15] Haket, M. P.
Parametrized Hardware Oriented VHDL models for abstract datatypes.
Faculty of Electrical Engineering, Section of Digital Information Systems,
Eindhoven University Of Technology, August 1992.
Master's Thesis no. 5947

[16] Van Lieshout, L.P.M.
Development of an emulation flow as part of the PCALE Design Flow.
Master's Thesis.
Department of Electrical Engineering, Design Automation Section,
Eindhoven University Of Technology, February 1994.

[17] 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.

Appendix B. List of Figures

Appendix C. Framework of definitions package

```
-- definitions package framework; to be adapted by replacement tool
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

PACKAGE definitions IS

    -- constants for symbolic use
    CONSTANT numb_of_addr_bits    : INTEGER :=
    CONSTANT numb_of_words        : INTEGER :=
    CONSTANT ENABLED              : bit     :=
    CONSTANT responsetime         : TIME    :=

    CONSTANT registerlength       : INTEGER :=
    CONSTANT wordlength           : INTEGER :=

    CONSTANT READ                 : bit     :=
    CONSTANT WRITE                : bit     :=

    FUNCTION logictoint(x : std_ulogic_vector) RETURN INTEGER;        -- SIM-LINE

END definitions;

PACKAGE BODY definitions IS                                           -- SIM-LINE

    FUNCTION logictoint(x : std_ulogic_vector) RETURN INTEGER IS      -- SIM-LINE
        VARIABLE y : INTEGER;                                         -- SIM-LINE
    BEGIN                                                             -- SIM-LINE
        y := 0;                                                       -- SIM-LINE
        FOR i IN x'LENGTH-1 DOWNTO 0 LOOP                             -- SIM-LINE
            IF x(i) = '1' THEN                                        -- SIM-LINE
                y := y + 2**i;                                        -- SIM-LINE
            END IF;                                                   -- SIM-LINE
        END LOOP;                                                     -- SIM-LINE
        RETURN y;                                                     -- SIM-LINE
    END logictoint;                                                   -- SIM-LINE

END definitions;                                                      -- SIM-LINE
```

Appendix D. Example of definitions package

```
-- definitions package; created by replacement tool
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

PACKAGE definitions IS

    -- constants for symbolic use
    CONSTANT numb_of_addr_bits    : INTEGER  := 12;
    CONSTANT numb_of_words        : INTEGER  := 4096;
    CONSTANT ENABLED              : bit      := '0';
    CONSTANT responsetime         : TIME     := 25 ns;                    -- SIM-LINE

    CONSTANT registerlength       : INTEGER  := 16;
    CONSTANT wordlength           : INTEGER  := 4;

    CONSTANT READ                 : bit      := '1';
    CONSTANT WRITE                : bit      := '0';

    FUNCTION logictoint(x : std_ulogic_vector) RETURN INTEGER;             -- SIM-LINE

END definitions;

PACKAGE BODY definitions IS                                                -- SIM-LINE

FUNCTION logictoint(x : std_ulogic_vector) RETURN INTEGER IS               -- SIM-LINE
    VARIABLE y : INTEGER;                                                  -- SIM-LINE
BEGIN                                                                      -- SIM-LINE
    y := 0;                                                                -- SIM-LINE
    FOR i IN x'LENGTH-1 DOWNTO 0 LOOP                                      -- SIM-LINE
        IF x(i) = '1' THEN                                                 -- SIM-LINE
            y := y + 2**i;                                                 -- SIM-LINE
        END IF;                                                            -- SIM-LINE
    END LOOP;                                                              -- SIM-LINE
    RETURN y;                                                              -- SIM-LINE
END logictoint;                                                            -- SIM-LINE

END definitions;                                                           -- SIM-LINE
```

Appendix E. VHDL model of memory

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.definitions.ALL;

ENTITY memory IS
    PORT( address       : IN std_ulogic_vector(numb_of_addr_bits-1 DOWNTO 0);
          data_ram_in   : IN std_ulogic_vector(wordlength-1 DOWNTO 0);
          r_w           : IN bit;
          ce            : IN bit;
          data_ram_out  : OUT std_ulogic_vector(wordlength-1 DOWNTO 0)
        );
END memory;

ARCHITECTURE registerlevel OF memory IS
    SUBTYPE dataword_type IS std_ulogic_vector(wordlength-1 DOWNTO 0);
    TYPE memory_type IS ARRAY(0 TO numb_of_words-1) OF dataword_type;
    SIGNAL mem : memory_type;
BEGIN

    operation : PROCESS(address)
    BEGIN

      IF address'EVENT THEN          -- RAM has to perform new action

        IF r_w = WRITE THEN          -- write operation
          mem(logictoint(address)) <= data_ram_in AFTER responsetime;
        ELSE                         -- read operation
          data_ram_out <= mem(logictoint(address)) AFTER responsetime;
        END IF;

      END IF;

    END PROCESS;

END registerlevel;
```

Appendix F. VHDL models of addressgenerator

This appendix contains two examples of the VHDL model of the addressgenerator. The first example is of a one-dimensional register, the second example is of a two-dimensional register.

Example of VHDL model of addressgenerator for a one-dimensional register

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.definitions.ALL;
USE work.exemplar_1164.ALL;

ENTITY addr_gen IS
    PORT ( address_in     : IN INTEGER RANGE 0 TO registerlength-1;
           address_out    : OUT std_ulogic_vector(numb_of_addr_bits-1 DOWNTO 0);
           ce             : OUT bit
         );
END addr_gen;

ARCHITECTURE registerlevel OF addr_gen IS
BEGIN

    -- RAM is always enabled
    ce <= ENABLED;

    -- the conversion function int2evec is defined in the exemplar_1164 package and is exemplar specific
    conv_addr : PROCESS(address_in)
        VARIABLE temp_var : std_ulogic_vector(3 DOWNTO 0);
    BEGIN
        temp_var := int2evec(address_in,4);
        address_out(1 DOWNTO 0) <= temp_var(3 DOWNTO 2);
        address_out(numb_of_addr_bits-1 DOWNTO 2) <= "0000000000";
    END PROCESS;

END registerlevel;
```

Example of VHDL model of addressgenerator for a two-dimensional register

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.definitions.ALL;
USE work.exemplar_1164.ALL;

ENTITY addr_gen IS
    PORT ( address_in     : IN INTEGER RANGE 0 TO registerlength-1;
           address_out    : OUT std_ulogic_vector(numb_of_addr_bits-1 DOWNTO 0);
           ce             : OUT bit
         );
END addr_gen;

ARCHITECTURE registerlevel OF addr_gen IS
BEGIN

    -- RAM is always enabled
    ce <= ENABLED;

    -- the conversion function int2evec is defined in the exemplar_1164 package and is exemplar specific
    conv_addr : PROCESS(address_in)
    BEGIN
        address_out(5 DOWNTO 0) <= int2evec(address_in,6);
        address_out(numb_of_addr_bits-1 DOWNTO 6) <= "000000";
    END PROCESS;

END registerlevel;
```

Appendix G. Listing of simple testcase

This appendix contains three VHDL descriptions of the simple testcase. The first description is the testcase before register replacement has been performed. The other two descriptions are the testcase after the replacement: the first description is used for simulation and the second description is used for synthesis.

Testcase before register replacement

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.definitions.ALL;

ENTITY dsn IS
    PORT(clk          : IN std_ulogic;
         rst          : IN std_ulogic;
         data_read    : OUT std_ulogic_vector(wordlength-1 DOWNTO 0)
        );
END dsn;

ARCHITECTURE dsnreg OF dsn IS
BEGIN
    regpcs : PROCESS(clk)
      VARIABLE index       : INTEGER RANGE 0 TO registerlength-1;
      VARIABLE repl_reg    : std_ulogic_vector(registerlength-1 DOWNTO 0);
      TYPE state_type IS   (RST,ONE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,NINE,
                            TEN,ELEVEN,TWELVE,THIRTEEN,FOURTEEN,FIFTEEN);
      VARIABLE state       : state_type;
    BEGIN
      IF clk'EVENT AND clk = '1' THEN

        IF rst = '1' THEN
          state := RST;
        END IF;

        CASE state IS
          WHEN RST      => state := ONE;
          WHEN ONE      => -- writing of first data
                           repl_reg(index+wordlength-1 DOWNTO index) := "0001";
                           index := index+wordlength;
                           state <= TWO;
          WHEN TWO      => -- writing of second data
                           repl_reg(index+wordlength-1 DOWNTO index) := "0010";
                           index := index+wordlength;
                           state <= THREE;
          WHEN THREE    => -- writing of third data
                           repl_reg(index+wordlength-1 DOWNTO index) := "0100";
```

```
                    index := index+wordlength;
                    state <= FOUR;
        WHEN FOUR   => -- writing of fourth data
                    repl_reg(index+wordlength-1 DOWNTO index) := "1000";
                    index := 0;
                    state<= FIVE;
        WHEN FIVE   => state <= SIX;
        WHEN SIX    => state <= SEVEN;
        WHEN SEVEN  => state <= EIGHT;
        WHEN EIGHT  => state <= NINE;
        WHEN NINE   => state <= TEN;
        WHEN TEN    => state <= ELEVEN;
        WHEN ELEVEN => -- reading of first data
                    data_read <= repl_reg(index+wordlength-1 DOWNTO index);
                    index := index+wordlength;
                    state <= TWELVE;
        WHEN TWELVE => -- reading of second data
                    data_read <= repl_reg(index+wordlength-1 DOWNTO index);
                    index := index+wordlength;
                    state <= THIRTEEN;
        WHEN THIRTEEN => -- reading of third data
                    data_read <= repl_reg(index+wordlength-1 DOWNTO index);
                    index := index+wordlength;
                    state <= FOURTEEN;
        WHEN FOURTEEN => -- reading of fourth data
                    data_read <= repl_reg(index+wordlength-1 DOWNTO index);
                    index := 0;
                    state <= FIFTEEN;
        WHEN FIFTEEN => state <= ONE;
      END CASE;
    END IF; -- RISING_EDGE(clk)
  END PROCESS; -- regpcs
END dsnreg;
```

Testcase for simulation after replacement

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.definitions.ALL;

ENTITY dsn IS
  PORT (clk       : IN std_ulogic;
        rst       : IN std_ulogic;
        data_read : OUT std_ulogic_vector(wordlength-1 DOWNTO 0)
       );
END dsn;
```

```
ARCHITECTURE dsnram OF dsn IS
    COMPONENT memory
      PORT (address          : IN std_ulogic_vector(numb_of_addr_bits-1 DOWNTO 0);
            data_ram_in      : IN std_ulogic_vector(wordlength-1 DOWNTO 0);
            r_w              : IN bit;
            ce               : IN bit;
            data_ram_out     : OUT std_ulogic_vector(wordlength-1 DOWNTO 0)
           );
    END COMPONENT;
    COMPONENT addr_gen
      PORT (address_in       : IN INTEGER RANGE 0 TO registerlength-1;
            address_out      : OUT std_ulogic_vector(numb_of_addr_bits-1 DOWNTO 0);
            ce               : OUT bit
           );
    END COMPONENT;
    FOR ALL : memory   USE ENTITY work.memory;
    FOR ALL : addr_gen USE ENTITY work.addr_gen;

    SIGNAL address           : INTEGER RANGE 0 TO registerlength-1;
    SIGNAL data_ram_in       : std_ulogic_vector(wordlength-1 DOWNTO 0);
    SIGNAL r_w               : bit;
    SIGNAL data_ram_out      : std_ulogic_vector(wordlength-1 DOWNTO 0);
    SIGNAL ce                : bit;
    SIGNAL address_to_ram    : std_ulogic_vector(numb_of_addr_bits-1 DOWNTO 0);

BEGIN
    -- send address for conversion to address_generator
    ag1 : addr_gen
    PORT MAP(address,address_to_ram,ce);

    -- send data to and receive data from memory
    ram1 : memory
    PORT MAP(address_to_ram,data_ram_in,r_w,ce,data_ram_out);

    rampcs : PROCESS(clk)
      VARIABLE index       : INTEGER RANGE 0 TO registerlength-1;
      VARIABLE sample_reg  : std_ulogic_vector(wordlength-1 DOWNTO 0);
      TYPE state_type IS   (RST,ONE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,NINE,
                            TEN,ELEVEN,TWELVE,THIRTEEN,FOURTEEN,FIFTEEN);
      VARIABLE state       : state_type;
    BEGIN
      IF clk'EVENT AND clk = '1' THEN

        r_w <= READ;
        sample_reg := data_ram_out;
```

```
IF rst = '1' THEN
  state := RST;
END IF;

CASE state IS
   WHEN RST        => address <= registerlength-1;
                      r_w <= READ;
                      state := ONE;
   WHEN ONE        => -- writing of first data
                      data_ram_in <= "0001";
                      index := 0;
                      address <= index;
                      r_w <= WRITE;
                      state := TWO;
   WHEN TWO        => -- writing of second data
                      data_ram_in <= "0010";
                      index := index+wordlength;
                      address <= index;
                      r_w <= WRITE;
                      state := THREE;
   WHEN THREE      => -- writing of third data
                      data_ram_in <= "0100";
                      index := index+wordlength;
                      address <= index;
                      r_w <= WRITE;
                      state := FOUR;
   WHEN FOUR       => -- writing of fourth data
                      data_ram_in <= "1000";
                      index := index+wordlength;
                      address <= index;
                      r_w <= WRITE;
                      state := FIVE;
   WHEN FIVE       => state := SIX;
   WHEN SIX        => state := SEVEN;
   WHEN SEVEN      => state := EIGHT;
   WHEN EIGHT      => state := NINE;
   WHEN NINE       => state := TEN;
   WHEN TEN        => -- address for next read operation
                      index := 0;
                      address <= index;
                      state := ELEVEN;
   WHEN ELEVEN     => -- reading of first data
                      data_read <= sample_reg;
                      index := index+wordlength;
                      address <= index;
                      state := TWELVE;
```

```
            WHEN TWELVE     => -- reading of second data
                               data_read <= sample_reg;
                               index := index+wordlength;
                               address <= index;
                               state := THIRTEEN;
            WHEN THIRTEEN   => -- reading of third data
                               data_read <= sample_reg;
                               index := index+wordlength;
                               address <= index;
                               state := FOURTEEN;
            WHEN FOURTEEN   => -- reading of fourth data
                               data_read <= sample_reg;
                               state := FIFTEEN;
            WHEN FIFTEEN    => state := ONE;
          END CASE;
        END IF; -- RISING_EDGE(clk)
      END PROCESS; -- rampcs
    END dsnram;
```

Testcase for synthesis after replacement

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.definitions.ALL;

ENTITY dsn IS
    PORT(clk            : IN std_ulogic;
         rst            : IN std_ulogic;
         data_read      : OUT std_ulogic_vector(wordlength-1 DOWNTO 0);
         data_ram_out   : IN std_ulogic_vector(wordlength-1 DOWNTO 0);
         data_ram_in    : OUT std_ulogic_vector(wordlength-1 DOWNTO 0);
         address_to_ram : OUT std_ulogic_vector(numb_of_addr_bits-1 DOWNTO 0);
         r_w            : OUT bit;
         ce             : OUT bit
        );
END dsn;

ARCHITECTURE dsnram OF dsn IS
--  COMPONENT memory
--    PORT (address           : IN std_ulogic_vector(numb_of_addr_bits-1 DOWNTO 0);
--          data_ram_in       : IN std_ulogic_vector(wordlength-1 DOWNTO 0);
--          r_w               : IN bit;
--          ce                : IN bit;
--          data_ram_out      : OUT std_ulogic_vector(wordlength-1 DOWNTO 0)
--         );
--  END COMPONENT;
```

```
    COMPONENT addr_gen
      PORT (address_in          : IN INTEGER RANGE 0 TO registerlength-1;
            address_out         : OUT std_ulogic_vector(numb_of_addr_bits-1 DOWNTO 0);
            ce                  : OUT bit
           );
    END COMPONENT;
--  FOR ALL : memory   USE ENTITY work.memory;
    FOR ALL : addr_gen  USE ENTITY work.addr_gen;

--  SIGNAL address          : INTEGER RANGE 0 TO registerlength-1;
--  SIGNAL data_ram_in      : std_ulogic_vector(wordlength-1 DOWNTO 0);
--  SIGNAL r_w              : bit;
--  SIGNAL data_ram_out     : std_ulogic_vector(wordlength-1 DOWNTO 0);
--  SIGNAL ce               : bit;
--  SIGNAL address_to_ram   : std_ulogic_vector(numb_of_addr_bits-1 DOWNTO 0);

BEGIN
    -- send address for conversion to address_generator
    ag1 : addr_gen
    PORT MAP(address,address_to_ram,ce);

    -- send data to and receive data from memory
--  ram1 : memory
--  PORT MAP(address_to_ram,data_ram_in,r_w,ce,data_ram_out);

    rampcs : PROCESS(clk)
      VARIABLE index        : INTEGER RANGE 0 TO registerlength-1;
      VARIABLE sample_reg   : std_ulogic_vector(wordlength-1 DOWNTO 0);
      TYPE state_type IS    (RST,ONE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,NINE,
                             TEN,ELEVEN,TWELVE,THIRTEEN,FOURTEEN,FIFTEEN);
      VARIABLE state        : state_type;
    BEGIN
      IF clk'EVENT AND clk = '1' THEN

        r_w <= READ;
        sample_reg := data_ram_out;

        IF rst = '1' THEN
          state := RST;
        END IF;

        CASE state IS
          WHEN RST      => address <= registerlength-1;
                           r_w <= READ;
                           state := ONE;
          WHEN ONE      => -- writing of first data
                           data_ram_in <= "0001";
```

```
                    index := 0;
                    address <= index;
                    r_w <= WRITE;
                    state := TWO;
WHEN TWO        => -- writing of second data
                    data_ram_in <= "0010";
                    index := index+wordlength;
                    address <= index;
                    r_w <= WRITE;
                    state := THREE;
WHEN THREE      => -- writing of third data
                    data_ram_in <= "0100";
                    index := index+wordlength;
                    address <= index;
                    r_w <= WRITE;
                    state := FOUR;
WHEN FOUR       => -- writing of fourth data
                    data_ram_in <= "1000";
                    index := index+wordlength;
                    address <= index;
                    r_w <= WRITE;
                    state := FIVE;
WHEN FIVE       => state := SIX;
WHEN SIX        => state := SEVEN;
WHEN SEVEN      => state := EIGHT;
WHEN EIGHT      => state := NINE;
WHEN NINE       => state := TEN;
WHEN TEN        => -- address for next read operation
                    index := 0;
                    address <= index;
                    state := ELEVEN;
WHEN ELEVEN     => -- reading of first data
                    data_read <= sample_reg;
                    index := index+wordlength;
                    address <= index;
                    state := TWELVE;
WHEN TWELVE     => -- reading of second data
                    data_read <= sample_reg;
                    index := index+wordlength;
                    address <= index;
                    state := THIRTEEN;
WHEN THIRTEEN   => -- reading of third data
                    data_read <= sample_reg;
                    index := index+wordlength;
                    address <= index;
                    state := FOURTEEN;
```

```
            WHEN FOURTEEN  => -- reading of fourth data
                              data_read <= sample_reg;
                              state := FIFTEEN;
            WHEN FIFTEEN      => state := ONE;
        END CASE;
    END IF; -- RISING_EDGE(clk)
END PROCESS; -- rampcs
END dsnram;
```

Appendix H. Example of control file

The necessary parameters, mandatory and optional, are passed on to the replacement tool by means of a control file.

Below an example of such a control file is given. The replacement tool considers all lines starting with a #-symbol as comment lines. Any number of comment lines and empty lines is allowed; they are ignored by the replacement tool. The syntax for non-comment, non-empty lines is: <parameter> <value>.

```
#####################################################
# MANDATORY PARAMETERS; MUST BE SPECIFIED  #
#####################################################
# The clock frequency (in MHz) on which the total design must function
CLOCK_FREQUENCY                         30

# The register that is being replaced
REGISTER_TO_REPLACE                     priv_segm_reg

# Length of the above register
REGISTER_LENGTH                         16000

# Variable that is used for indexing the register that is being replaced
INDEX_VARIABLE                          write_pointer

# Length of the above variable
INDEX_LENGTH                            8

# String to use for detection of active edge
ACTIVE_EDGE_MARK                        IF (clk'EVENT AND clk = '1') THEN

#############################################################
# MANDATORY PARAMETER FOR TWO-DIMENSIONAL REGISTERS  #
#############################################################
# Name of type declaration in case of a two-dimensional register
2D_TYPE                                 reg_type


#####################################################
# OPTIONAL PARAMETERS; NEED NOT BE SPECIFIED  #
#####################################################
# Name of the RAM, present in the RAM library, to use for replacement
RAM_NAME                                CY7C171

# Extension for output simulation files; default = ".sim.vhdl"
SIMULATION_FILES_EXTENSION              .sim.vhdl

# Extension for output synthesis files; default = ".syn.vhdl"
SYNTHESIS_FILES_EXTENSION               .syn.vhdl

# Extension for temporary result files; default = ".r2r"
TEMPORARY_FILES_EXTENSION               .r2r
```

```
# Extension for temporary files containing procedure bodies; default = ".pbf"
PROCEDURE_BODY_FILES_EXTENSION          .pbf

# Name of package file containing constants, etcetera; default = "definitions.vhdl"
PACKAGE_FILE                            definitions.vhdl

# Name of file containing framework for package file; default = "definitions.vhdl.frame"
PACKAGE_FRAME_FILE                      definitions.vhdl.frame

# Name of file containing addressgenerator; default = "ram.vhdl"
ADDR_GEN_FILE                           ram.vhdl

# Name of file containing framework for addressgenerator; default = "ram.vhdl.frame"
ADDR_GEN_FRAME_FILE                     ram.vhdl.frame

# Name of file containing RAMs to select from; default = "ram.library"
RAM_LIBRARY_FILE                        ram.library

# Name of file to write transcript to; default = "transcript.r2r"
TRANSCRIPT_FILE                         transcript.r2r

# Name of constant indicating length of words of RAM; default = "wordlength"
WORDLENGTH_CONSTANT_NAME                wordlength

# Name of constant indicating number of address bits of RAM;
# default = "numb_of_addr_bits"
NUMB_OF_ADDR_BITS_CONSTANT_NAME         numb_of_addr_bits

# Name of constant indicating number of words of RAM; default = "numb_of_words"
NUMB_OF_WORDS_CONSTANT_NAME             numb_of_words

# Name of constant indicating registerlength; default = "registerlength"
REGISTERLENGTH_CONSTANT_NAME            registerlength

# Name of sampling register that is inserted; default = "sample_reg"
SAMPLING_REGISTER_NAME                  sample_reg

# Name of address signal as input for addressgenerator that is inserted; default = "address"
ADDRESS_TO_ADDR_GEN_SIGNAL_NAME         address

# Name of data_ram_in signal that is inserted; default = "data_ram_in"
DATA_RAM_IN_SIGNAL_NAME                 data_ram_in

# Name of read_write signal that is inserted; default = "r_w"
READ_WRITE_SIGNAL_NAME                  r_w

# Name of data_ram_out signal that is inserted; default = "data_ram_out"
DATA_RAM_OUT_SIGNAL_NAME                data_ram_out

# Name of RAM enable signal that is inserted; default = "ce"
RAM_ENABLE_SIGNAL_NAME                  ce

# Name of address_to_ram signal that is inserted; default = "address_to_ram"
ADDRESS_TO_RAM_SIGNAL_NAME              address_to_ram
```

Appendix I. Example of files file

Since a design can be distributed over multiple files, all these files have to be known to the replacement tool. Therefore all files have to be stated in a "files file".

Below an example of such a files file is given. The replacement tool considers all lines starting with a #-symbol as comment lines. Any number of comment lines and empty lines is allowed; they are ignored by the replacement tool. The syntax for non-comment, non-empty lines is: <file name>.

```
# This file defines all the files to consider when converting a register to a RAM

# packages
demux_pack.vhdl

# I/O routines
demux_data_io.vhdl

# demux parser
parser_wd930612.vhdl
```

Appendix J. Example of RAM library

Every replacement may require its own specific RAM. All characteristics of the RAM that is used for the replacement have to be known to the replacement tool. The most flexible solution is to state all characteristics of all possible RAMs in a separate file: a RAM library.

Below an example of such a RAM library is given. The replacement tool considers all lines starting with a #-symbol as comment lines. Any number of comment lines and empty lines is allowed; they are ignored by the replacement tool. The syntax for non-comment, non-empty lines is: <characteristic> <value>.

```
# This file is a RAM library

RAM_NAME                CY7C171
    WORD_LENGTH         4
    NUMB_OF_ADDR_BITS   12
    NUMB_OF_WORDS       4096
    ENABLE_VALUE        0
    RESPONSE_TIME       25
    READ_VALUE          1
    WRITE_VALUE         0
    STATUS              AVAILABLE

# Note that the order of stating the characteristics can be changed

RAM_NAME                CY7C167
    STATUS              NOT AVAILABLE
    RESPONSE_TIME       25
    READ_VALUE          1
    WRITE_VALUE         0
    ENABLE_VALUE        0
    WORD_LENGTH         1
    NUMB_OF_WORDS       16384
    NUMB_OF_ADDR_BITS   14
```

Appendix K. Glossary

architecture	- defines the relationships between input and outputs of an entity
ASIC	- Application Specific Integrated Circuit
benchmark	- well-known and well-defined design used as standard testcase; comparison between software tools, hardware modules etcetera, is usually based on benchmarks
bit	- binary digit
BRD	- Bandwidth Restoration Decoder, decoder in an HD-MAC receiver
bread board	- circuit board for system development and testing
byte	- group of bits (usually eight)
constraint	- limitation of possible values
COST	- function for partitioning hardware and software, taking into account Customer requirements, Overall development cost, Silicon area & package and Time-To-Market
CRC	- Cyclic Redundancy Check, error detecting code
DSA	- Design Style Assistant, tool developed for bridging the gap between an HL description of an IC and synthesis tools as part of the Advanced PCALE Design Flow, subject of this Master's Thesis
DVB	- Digital Video Broadcasting
DVP	- Digital Video Processing, design group at PCALE
EDIF	- Electronic Design Interchange Format
emulation	- design functionality check by means of hardware
entity	- primary hardware abstraction in VHDL
EPLD	- Erasable Programmable Logic Device
FIFO	- First In First Out
flip-flop	- hardware element, capable of retaining one logic value
FDM	- Frequency Division Multiplexing, multiplexing on basis of different frequencies
function	- VHDL construct, used for abstraction of an algorithm (or part of it) to a single expression
gate array	- half-fabricated ICs: the logic cells are already fabricated but the interconnections (wiring) still have to be made through two final IC masks
gate level	- logic gate description level
hardware	- circuit board(s)
HD	- High Definition

HDL	- Hardware Description Language
HD-MAC	- High Definition Multiplexed Analogue Components, compatible HDTV standard developed in the European Eureka_95 project
HDTV	- High Definition Television
HL	- High Level description of a design
Hz	- Hertz, unit of frequency (s^{-1})
IC	- Integrated Circuit (chip)
IEEE	- Institute of Electrical and Electronics Engineers
I/O	- Input/Output
ISO	- International Standards Organization
library	- collection of similar objects
LL	- Library Level description of a design
µC	- micro-controller
Mbit	- 10^6 bit
MHz	- 10^6 Hz
ML	- Medium Level description of a design
modulation	- shifting information to a higher frequency to improve transmission
MPEG	- Moving Pictures Expert Group, standard for compression of digital image data
MPEG-1	- version 1 of the MPEG-standard, digital image compression algorithm originally intended for digital storage media, capable of reproducing full motion video at bit rates of about 1.5 Mbit/s
MPEG-2	- version 2 of the MPEG-standard, extension to version 1 in the sense that it enables full motion image reproduction at bit rates up to and including 15 Mbit/s, aimed at digital TV broadcast systems
multiplexing	- mixing of different data signals into one signal
netlist	- listing of a gate level implementation of an IC, containing gates and interconnection
package	- VHDL construct, provides a means of defining subprograms and other resources in a way that allows different design units to share the same declarations; also the packing of an IC
PCALE	- Product Concept and Application Laboratory of Philips Semiconductors in Eindhoven
PLD	- Programmable Logic Device
PREP	- Programmable Electronic Performance Corporation, a consortium of 13 prominent suppliers of programmable logic and tools

procedure	- VHDL construct, used for abstraction of an algorithm (or part of it) to a single statement
RAM	- Random Access Memory
register	- group of flip-flops, used for retaining some logic values
RTL	- Register Transfer Level, describing a design at this level means that the complete structure of the design is described
SIL	- Sprite Input Language, an intermediate language between high level specification languages and synthesis tools
simulation	- design functionality check by means of software
software	- computer program(s)
subprogram	- procedure or function as part of a VHDL description, defines algorithm for computing values or exhibiting behaviour
synthesis	- creation of an implementation of an IC from a description of an IC
synthesis library	- library containing all primitive building blocks that can be used by a synthesis tool to synthesize a design description
synthesis tool	- software used for (semi-)automatic creation of an implementation of an IC from a description of an IC
TDM	- Time Division Multiplexing, multiplexing on basis of time sharing
template	- prescribed framework
tool	- software for performing a specific task
TUE	- Eindhoven University of Technology
TV	- Television
TXT	- Teletext
VHDL	- VHSIC Hardware Description Language
VHSIC	- Very High Speed Integrated Circuit
VLSI	- Very Large Scale Integration