Exploration des liens entre la synthèse de haut niveau (HLS) et la synthèse au niveau transferts de registres (RTL)

V. Vijayaraghavan

THESE

présentée par

Vijay P. VIJAYARAGHAVAN

pour obtenir le titre de **DOCTEUR**

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(arrêté ministérial du 30 Mars 1992)

Spécialité: **Microélectronique**

EXPLORATION DES LIENS ENTRE LA SYNTHESE DE HAUT NIVEAU (HLS) ET LA SYNTHESE AU NIVEAU TRANSFERTS DE REGISTRES (RTL)

Date de soutenance : **29 Novembre 1996**

Composition du jury : Messieurs

Bernard	COURTOIS	:	Président
Eric	MARTIN	:	Rapporteur
Jean-Pierre	SCHOELLKOPF	:	Rapporteur
Ahmed Amine	JERRAYA	:	Directeur

Thèse préparée au sein du Laboratoire TIMA-INPG
46, Avenue Félix Viallet, 38031 Grenoble, France.

1.1 Prologue

Ever since the introduction of the Metal Oxide Semiconductor (MOS), every 3 years a new technology generation has been emerging with 4 times the transistor count of the previous generation [Jac96]. Such an impressive progress in the micro electronics technology over the last two decades is expected to continue at least until the year 2010. *Figure* 1.1 shows the expected increase in complexity (number of transistors in millions) and a decrease in the technology minimum feature size (micro metres) for every 3 years:

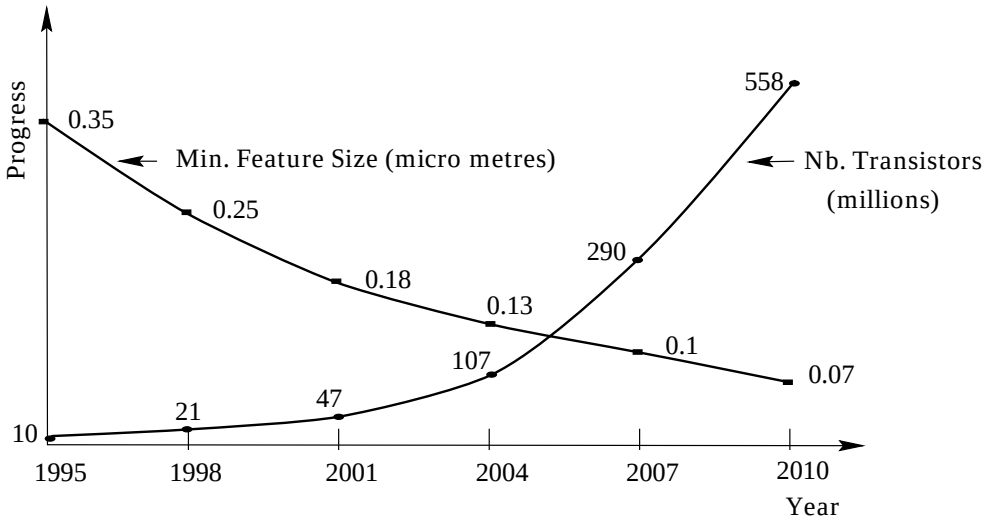


Figure 1.1: Progress in MOS integration

The main reasons for such a progress is both due to the advances in the process technology and the Electronic Design Automation (EDA) methodologies. We address the latter topic and in particular the High Level Synthesis (HLS). The HLS of a digital system is the realization of a RTL description starting from a High Level description. The RTL structure produced by a HLS tool is a set of interconnected components that realize the specified behaviour of the system. The RTL structure includes a controller and a datapath. The controller is a hardware synthesized to be a Finite State Machine (FSM). The datapath includes the Functional Units (FUs) such as the ADDers, MULTipliers, ALU etc, Storage Units (SUs) such as a Memory or a Register, the Communication Units (CUs) such as the BUS, Mux etc, and the Synchronization Units such as a Clock generator. Figure 1.2 shows a small example of HLS [Cam90] used to compute the factorial of a given input integer.

Nevertheless, in real life, a high level specification may not be able to be synthesized completely; It can be a mixture of synthesizable and some un-synthesizable specifications. The synthesizable specification is transformed into a register transfer level, while the un-synthesizable part is simply migrated to a lower level. In this thesis

work, we shall investigate the context of a HLS within the existing Computer Aided Design (CAD) methodologies at the Register Transfer Level (RTL).

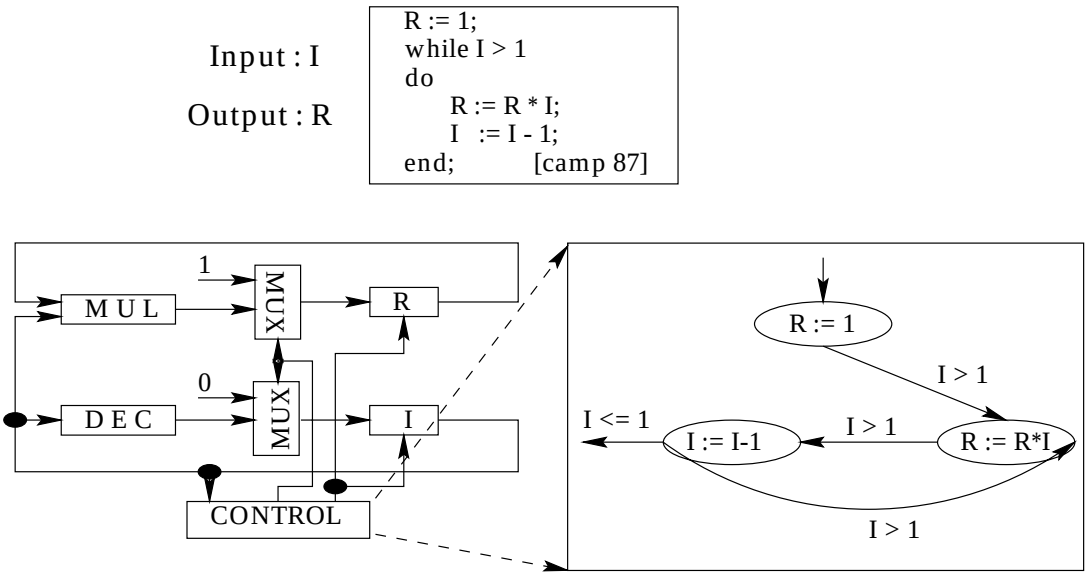


Figure 1.2: An example of HLS

1.2 Motivations

Designers willing to migrate the initial specification to a higher level of abstraction (in our case, from the RTL to Behavior) has the following requirements in mind:

1. Can I use the result of High Level Synthesis directly in my existing environement and is my design continue to be portable?
2. How do I reuse some of my existing lower level designs for the efficiency and Time-to-Market advantages?
3. What are my additional overheads in terms of learning and verification of my future designs and are they reasonable?

Those requirements become the motivations of our thesis work.

1.3 Problem Positioning

A classical HLS starts from a high level behavioral specification and then generates a RTL architecture containing a Controller and a Datapath as shown by the *Figure* 1.3 (a). This approach is well suited to high level specifications which are completely

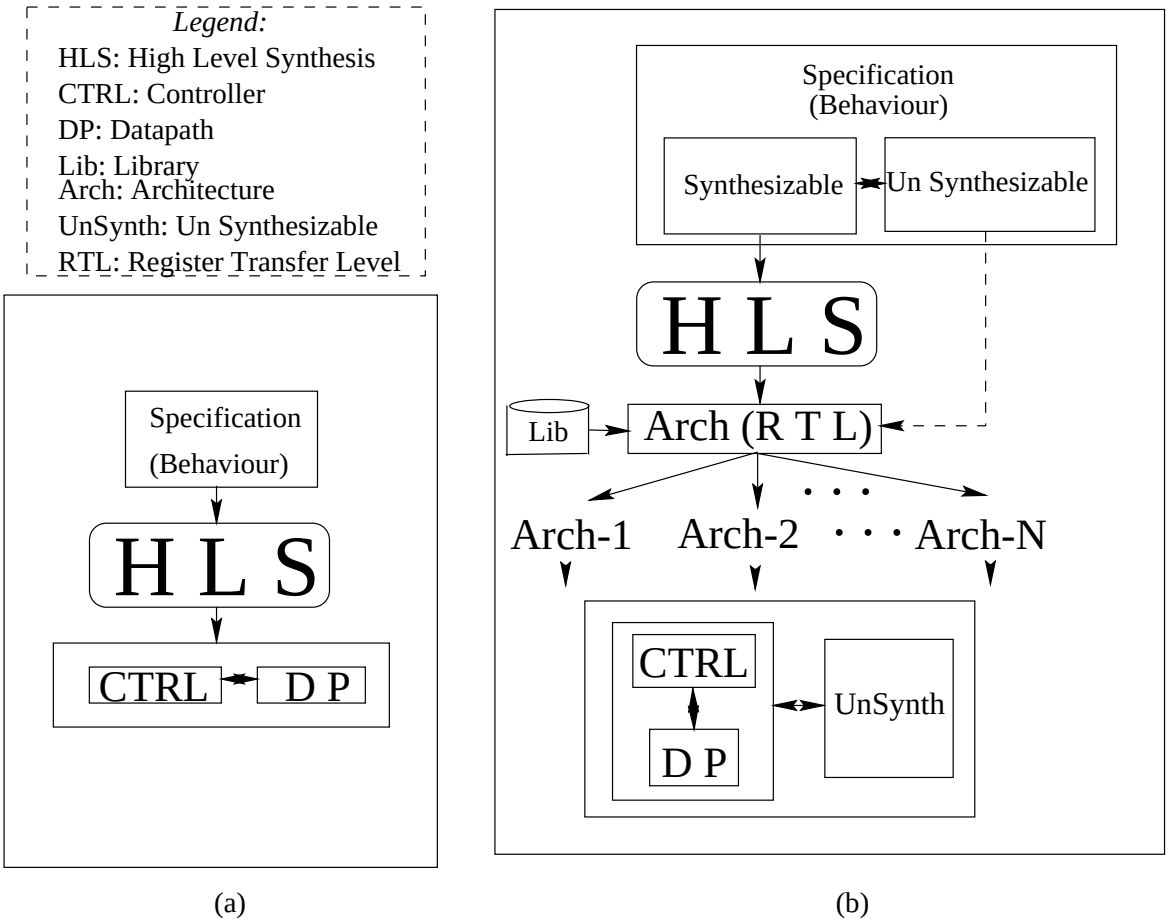


Figure 1.3: a) classical vs. b) flexible HLS

synthesizable. In fact, most of the HLS tools accept only a small subset of the specification languages. This subset is combined with a particular coding style and called as the synthesizable subset. However, in real life, specifications may contain a mixture of synthesizable and un-synthesizable parts. Some of the factors for the presence of un-synthesizable part(s) within a specification are the following:

1. Limitation imposed by the HLS tool (example: no multiple processes).
2. Need for reusable macro blocks either to avail the advantage of 'Time-to-market' or 'Efficiency' or both.
3. Possibility of customizing the synchronization between the Controller and the datapath.
4. Exploration of the design space with different choices for the architecture.
5. Presence of some part(s) of the initial specification at the RT (Example: a FSM process) or even a lower level (a structural description).

HLS, in general, makes use of a library of components such as the registers, mux, adders etc. In order to make a HLS independent of the technology, a generic component library should be used.

Theoritically, most of the HLS tools produce a RTL model using a fixed style of architecture made of a single irregular datapath and a single controller. For efficiency reasons, we may need to reorganize this model in order to better interface with the lower level synthesis tools such as a regular datapath compiler.

1.4 Contributions

The contributions of this thesis work is to provide a flexible, efficient and parameterizable link between HLS and RTL synthesis while meeting the requirements of the designer mentioned in a previous section. We summarize the goals as to be able to:

1. mix synthesizable and un synthesizable descriptions in the initial high level specification. This is required to validate the entire functional specification as the initial design step.

2. interface the result of high level synthesis efficiently with lower level synthesis tools.

3. handle a library of generic components.

These objectives are achieved respectively using the following three operations:

1 We define *Architecture Personalization* as a method to migrate unsynthesized parts of a specification from a High Level into RT level. Other aspects of personalization are considered in chapter-4.

2 We define *Architecture Decomposition* as a method that permits to extract Regular datapaths contained within a high level synthesized architecture. The extracted regular datapath(s) can be imported by a standard datapath compiler. A datapath compiler offers the advantages of parameterization, regularity and a possible area/time saving and thus being efficient. This aspect of decomposition is detailed in Chapter-5.

3 Architecture Translations convert the architecture into standard formats of VHDL and EDIF from an internal format called Solar [JO92]. The translated architecture is at the Register Transfer Level. This process handles abstract libraries. The translations and library mapping processes will be discussed in Chapter-6.

1.5 Chapter Preview

To round off this chapter, a brief preview of the contents of those still to come is now given. Each one deals with a particular problem area and is relatively self contained.

- Chapter-2 introduces to the High Level Synthesis presenting its State of the Art.
- In Chapter-3, we concentrate on the Amical HLS system which will be referred in the rest of our methodology.
- Chapter-4 describes the notion of a *Personalization* method suitable for the architecture obtained from the Amical system.
- Chapter-5 presents a method for the *Decomposition* of the datapath into regular datapaths in order to treat them more efficiently by a datapath compiler. The back-end library is also considered.
- Chapter-6 groups the procedures of the architecture translations from Solar into the standard formats of VHDL and EDIF within an Object Oriented environment.
- Finally, Chapter-7 concludes the thesis work with the contributions of our work to high level synthesis (in particular to the Amical system) and the avenues for future work.

.

Chapter 2

STATE OF THE ART : HLS

In this chapter, we introduce to the High Level (or) Algorithmic Synthesis. In order to cope with the increasing complexity of VLSI designs, several tools and languages have evolved during the past two decades in order to perform high level synthesis. High Level Synthesis (HLS) is a process that translates a specification at an algorithmic level into a Register Transfer Level (RTL) architecture for an eventual hardware realization. The success of a HLS tool very much depends on its integration into the existing CAD environment, which is the subject of this thesis work.

In this chapter, we present the state of the art of HLS studying some of the existing tools and then compare the way in which they interface with the lower level synthesis.

.

2.1 Introduction

High Level Synthesis (HLS) is a process to convert a specification at an algorithmic level into a hardware architecture for realizing it into a VLSI. It is also called as a behavioral synthesis or an algorithmic synthesis.

A lot of work has already been carried out in the domain of behavioral synthesis. Most well known scheduling and allocation techniques have been applied in high-level synthesis. The most used techniques are rule-based expert systems (DAA), greedy iterating algorithms (EMUCS CHARM, MABAL), branch and bound algorithms (MIMOLA [Zim79]), linear-programming algorithms (ADPS), clique-partitioning algorithms (FACET), simulated-annealing-based algorithms, path-based allocation algorithms and force-directed scheduling/allocation algorithms (HAL [PKG86]). More details about these techniques are given in [CB90, CT90, MCG$^+$90, DN89, ORJ93, RJ96].

Most of the existing high-level synthesis systems start from a data-flow graph representation of the design. This kind of representation is well suited for data-flow-based specifications, using applicative languages, for example. Only a small amount of research has been carried out into the data-path synthesis of control-flow-based systems. A few of the early silicon compilers tackled this problem with only a small amount of success. However, this domain is becoming popular because, after the success of the research into DSP systems, controller design is now the bottleneck in system-level synthesis.

For the specification of control-flow dominated applications such as complex controllers, an imperative specification is generally used. This specification is more closely related to a state diagram rather than to a data-flow graph. In this case, translating the initial specification into a data-flow graph is an un-optimization step. The application of the classical synthesis schemes will produce poor results in the case of large examples .

The principal steps of a high level synthesis are the *Compilation, Scheduling, Allocation and Architecture generation*:

1 **Compilation:** The source HDL is converted into an internal representation,usually a dataflow graph and/or a control flow graph [RJ96, GR94] . The internal representation is then transformed into a form more suitable for the HLS. Those transformations involve both compiler-like and hardware-specific transformations.

2 **Scheduling:** Scheduling applied to synchronous digital systems consists of determining the control steps or states at which different operations are performed and is *NP*-complete. The control steps are equivalent to states in a control FSM, or microprogram steps in a microprogrammed controller. The operations are

such as addition, multiplication, logic operations, assignment etc. [Cam90]. Scheduling assigns each operation to a control step. The cost function to be optimized by the scheduling is a combination of the amount of hardware, the cycle time, and the number of control steps required. The amount of hardware is a measure which may include functional units such as ALUs, adders and simple gates, storage units such as memories and registers, and communication units such as buses and multiplexors etc.

Some of the well known scheduling techniques are the following [CB90, Cam91]:

- First Come First Served (**FCFS**) Scheduling: This method considers only data dependencies. The ready operations are planned in that order, in which they were supplied from the previous step.
- As Fast As Possible (**AFAP**) Scheduling: Under consideration of data dependencies, each of the operations is planned as early as possible.
- As Late As Possible (**ALAP**) Scheduling: Under consideration of data dependencies, all operations are planned by the latest possible time, which results from a certain maximum time limit.

Both ALAP and AFAP schedulings start out from unlimited resources.

A path based scheduling using an algorithm for *As Fast As Possible* (AFAP) scheduling may be found in [RJ96]. The scheduling may be usually constrained in 2 ways: In a resource constrained scheduling, the number of control steps are minimized for the given resources and in a time constrained scheduling, the resources are minimized for the given number of time steps [GR94].

3 **Allocation:** The allocation consists of the following tasks: Selection of components to be used in the RT level design; Binding a hardware structure to behavioral operators and variables and defining the target datapath architecture such as a bus-based or a mux-based. The allocation step assigns each operation to a piece of hardware. Allocation involves both the selection of the type and quantity of hardware modules from a library and mapping of each operation to the selected hardware. This step also performs the register and connection allocations. There exists three main types of hardware allocation algorithms .

1. *Constructive* : Allocates hardware while minimizing the final cost. The obtained solution is the best that can be reached by the algorithm.
2. *Iterative* : Generates new solutions, each one better than the previous one, starting from an initial solution.
3. *Iterative-Constructive* : A combination of the two types above. They create an initial solution of a certain quality with a constructive algorithm and then produce improvements over it. Each of those types of algorithms sets different requirements on the area estimations.

4 **Architecture Generation:** The architecture generation step produces usually the design in terms of components that can be passed to the RTL or Logic synthesis. The generated architecture consists of two principle components: a controller and a datapath. Controller is modelled either as a finite state machine (FSM) or as micro-programmed. A FSM is suitable for application specific architectures with a complexity of the order of 1000 states. A micro-programmed controller is suitable for re programmable architectures as well as for large controllers. Datapath is an interconnection of components such as the functional units, communication units, switching units and so on. The components are described either at a Register Transfer Level or as black boxes. The latter is used whenever a component reuse is desired.

2.2 State of the Art

We will limit our interest to five of the HLS tools, namely: Cathedral [SN91, RMVGG88], HIS [CBH+91], VSS, Synopsys Behavioural Compiler [KLMM95, Syn94] and Amical [JBD+95, KDB+95, JPO93, POJ92, Par92]. Of course, the litterature reports more than about 100 HLS tools [Aic94]. Several recent studies present more detailed study of these tools [Kis96, WC91]. We will address particularly the problem of linking of a HLS tool into the existing RTL synthesis environments.

2.3 Cathedral HLS system

The Cathedral-2/3 system is designed to support the high level synthesis of Digital Signal Processing (DSP) algorithms. Cathedral-2 is used for applications of medium complexity, while, Cathedral-3 supports applications of larger complexity.

The input to the Cathedral system is a data flow description in the Silage [SJKR91] hardware description language. The architecture synthesized by the Cathedral 2/3 consists of a controller and a datapath. The controller is micro programmed and the datapath consists of Execution Units (EUs) such as a RAM, ROM, ALU etc. in the case of Cathedral-2 and/or complex Application Specific Units (ASUs) in the case of Cathedral-3. The HLS steps used by Cathedral are explained through Figure 2.1.

Cathedral-2/3 is well interfaced with the commercial Cadence CAD environment. The scheduling and allocation algorithms permit automatic pipelining. The back annotation is facilitated by earlier evaluations of timing and area. Following are, however, some disadvantages:
The system requires the use of a non (universally) standard input language, Silage.

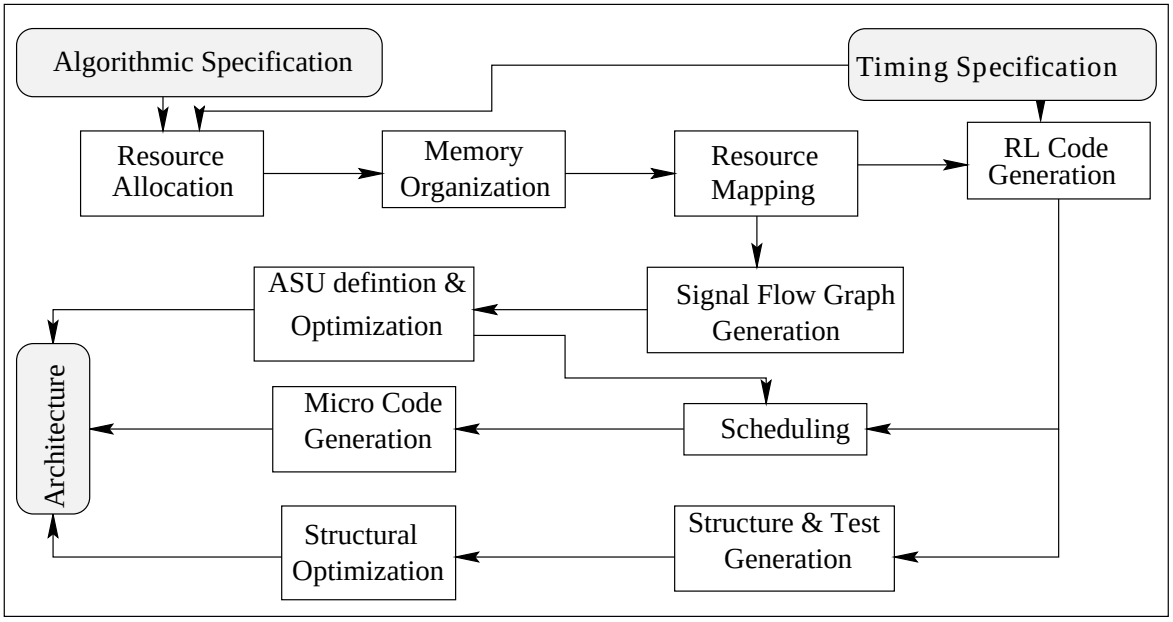


Figure 2.1: Cathedral High Level Synthesis Steps

The Execution Units are limited by a fixed execution time. The system requires 2 different simulation environments: Silage for the input and VHDL for the output. The conditional inputs (case) do not allow the use of a 'default' condition. This means that both 'true' and 'false' conditions must be explicitely specified.

2.3.1 Silage

Silage is a high level language optimized for specifying signal processing like algorithms. Silage is a data flow language that operates on streams of values. A silage program corresponds to a data flow graph – a directed graph with operations performed by vertices and values called edges. A subtraction node, for example, combines two streams of numbers to produce a different stream. In addition to the usual arithmetic functions, Silage provides some common DSP operations: delaying a stream Z^{-1}, taking a substream (decimation), and taking the union of streams (interpolation). The textual Silage program is a set of equations that defines output variables in terms of input variables. Variables can be subscripted in the usual way. The *iterated definition* provides a general way to operate on subscripted variables. Structural preferences are expressed by the *pragma* directives. Wang's master's thesis [Wan88] and the "Silage Reference Manual" specify the language in full. The Silage compiler translates a Silage program into *RL* to run on a single microcoded processor. The RL language is an approximate subset of C. It incorporates two major extensions: fixed-point types and register classes.

2.3.2 Interface with RTL synthesis

Cathedral performs a partial RTL synthesis and generates a netlist. It accepts only a synthesizable description and may be hardly interfaced with specific RTL synthesis tools such as a datapath compiler.

2.4 Synopsys Behavioural Compiler

The Synopsys behavioral compiler supports the high level synthesis of specifications containing both the data flow and control flow operations. The specifications are classified by the way they are scheduled; Three different scheduling modes are available:

- **Cycle fixed mode** of scheduling treats descriptions in which access on the inputs and outputs are described at a clock cycle level. This mode is also used in the VOTAN (previously, CALLAS) HLS system [BMK92] from the Siemens AG, Germany.

- **Superstate Fixed mode**

  This mode is similar to the scheduling step used in the INPG/TIMA's AMICAL HLS system. In this mode, the scheduling preserves the order of the input-output (I/O) operations. However, it may introduce new clock cycles between the successive behavioural operations, called as the *super state fixed* interval.

- **Free Floating mode**

  This mode is similar to the scheduling step employed by the IMEC's CATHEDRAL HLS system. In this mode, the scheduling is allowed to change the execution order of the I/O operations and to introduce new clock cycles. The design flow with the Synopsys Behavioral Compiler (BC) is explained in *Figure* 2.2.

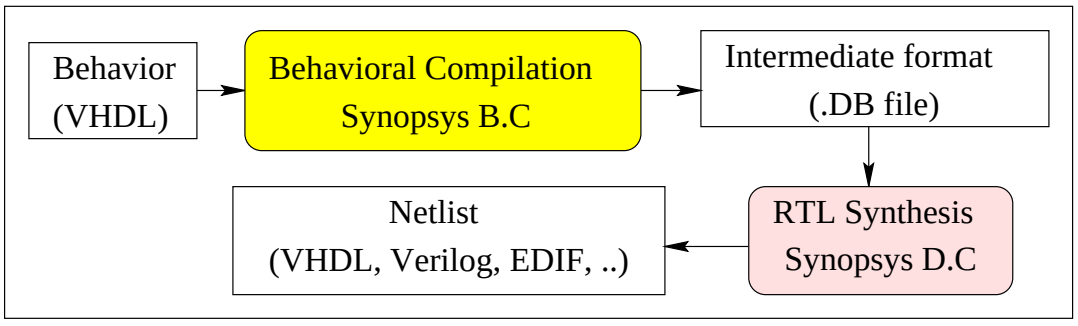


Figure 2.2: Synopsys Behavioral Compilation

2.4.1 Interface with the RTL synthesis

The BC produces an intermediate model that is directly used by the Synopsys' Design Compiler (DC) for RTL synthesis. Nevertheless Synopsys has its own RTL solutions, it is very hard to interface the result of BC with other RTL synthesis tools and methodologies.

2.5 HIS and VSS systems

The High Level Synthesis System (HIS) is from IBM's Thomas J.Watson Research center. It follows a similar approach as of the Synopsys' BC. It produces the synthesis results in an intermediate format for Booledozer [SKDB96], the in house logic synthesis tool of IBM. The VHDL Synthesis System (VSS) is an academic HLS tool developed at the University of California, Irvine. It follows the classical HLS flow. It uses a powerful library system called GENUS [Dut88] making the system independant of technology.

Table 2.1 summarizes the characteristics of the HLS tools in terms of the different aspects involved during linking the high level synthesis with the backend Register Transfer or Logic Level synthesis. The table also includes the relative characteristics of Amical.

Problem	Cathedral	HIS	VSS	Synopsys	Amical
Handling Different synchronization schemes	No	No	No	Yes	Yes
Handling Different libraries	Yes	No	Yes	Yes	Yes
Mixing Synthesizable and unsynthesizable descriptions	No	No	No	No/Yes	Yes
Handling Different RTL targets	Yes/No	No	Yes	No	Yes

Table 2.1: Comparison of linking issues between High Level and RTL

2.6 AMICAL HLS system : Objectives

AMICAL is a HLS system suited towards control flow dominated specifications. The system is explained in larger details in Chapter-3. The HLS steps used by Amical are explained through *Figure* 2.3.

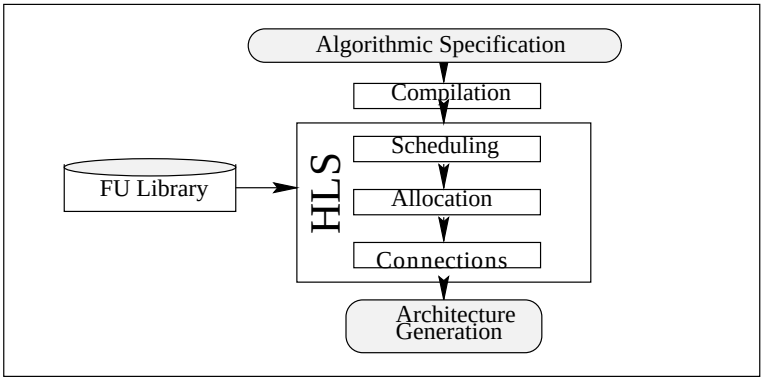


Figure 2.3: AMICAL High Level Synthesis Steps

The link between HLS and a Low Level Synthesis (LLS) in the context of Amical is shown in *Figure* 2.4

Our objective is to make this link more flexible and efficient. Recalling the other HLS tools presented above, we identify the following three problems:

1 The personalization of the synchronization scheme by the designers, the use of different libraries, mixing of unsynthesizable and synthesizable parts into the high level specifications are not flexibly handled by all the HLS tools.

2 Structural issues, such as the advantage of regularity, the level of primitives, and the influence of the size and the functionality of the partitions of combinational logic generated is not considered in HLS.

3 Many HLS tools have a considerable gap between HLS and the logic synthesis which needs to be bridged.

Amical solves the three above mentioned problems. The first issue is divided into 3 parts:

1. Handling of *different synchronization scheme* is supported by introducing an additional synchronization process together with a certain synchronization signals such as the clock(s), reset which are required at the RT level after a HLS.

2. Handling of *different libraries* is supported by the notion of pre defined and user defined conventions.

3. *Mixing* the Unsynthesized and synthesized parts of descriptions into an initial specification is enabled using a personalization file created by the user.

The second issue is handled by targetting the datapath output for different low level synthesis systems such as the logic synthesis, RTL synthesis, datapath compiler etc.

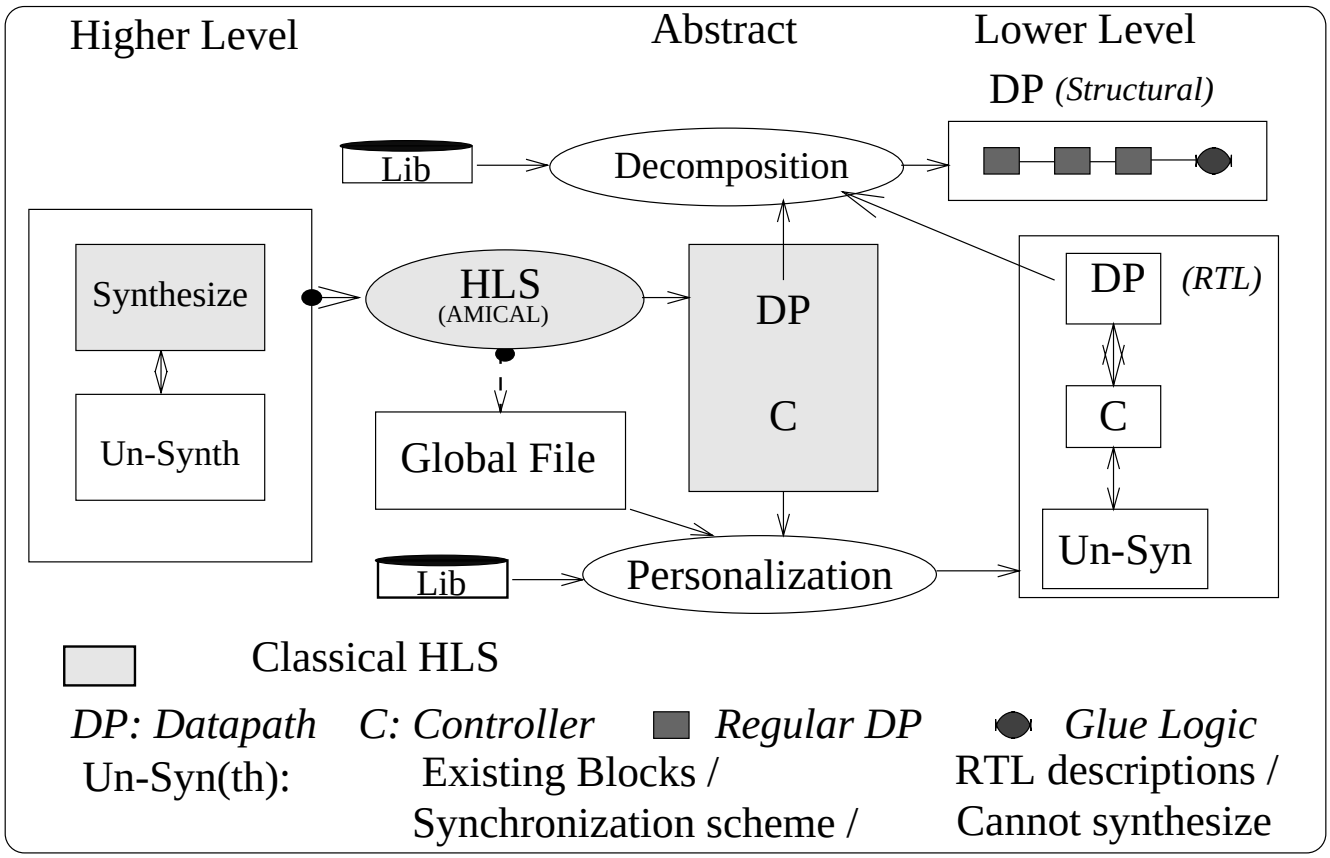


Figure 2.4: Linking HLS and LLS with AMICAL

The third issue is caused either because the output is not in a standard accepted format or unable to be validated by the simulation with respect to the initial specification. The first problem is solved by producing the output in the standard formats of VHDL and EDIF. The validation is ensured by allowing to use the same initial test bench.

Chapter 3

AMICAL

AMICAL[KDB+95, JPO93, POJ92, Par92] is a high level synthesis *system whose characterstics are the following:*
It is aimed towards control flow dominated machines [ORJ93, OPJC93, Wol91]. It interfaces well with the existing design environments and methodologies. It supports a standard interface: Starting with a pure VHDL[VHD87] input, AMICAL produces a full specification for the existing logic and RTL, the register transfer level synthesis tools [Aic94]. The target architecture is a synchronous, heterogeneous and parallel model. In order to handle very large designs, AMICAL allows the use of hierarchy and complex functional units [KDJ94]. It also allows the re use of existing components within new designs through the use of external library of macros. The combination of automatic and manual synthesis [Ding96] allows a quick and broad exploration of the design space in real time. The shorter response time of AMICAL, usually of the order of seconds, makes it a genuine interactive system. Several large examples have already been used for the evaluation with excellent results. These include a controller for a telephone answering machine [OPJC93], a motion estimator within a digital video codec [PFMH95, BKV+96], a speed controller system for controlling robatic motors [CRJ96] and a MPEG-AUDIO decoding system [KCBJ93]. All of the above examples are large designs whose individual synthesis time required only about 5 minutes.

.

3.1 Introduction

This chapter presents an advanced, high-level synthesis system called AMICAL that is targetted towards control-flow dominated machines. AMICAL starts with a functional specification given in VHDL and generates an architecture composed of a datapath and a controller that may feed existing synthesis tools acting at the logic and register-transfer levels. It also allows the re-use of existing components within the new designs using an external library of macros.

3.1.1 AMICAL: Motivations

Although behavioral silicon compilation has made large strides towards the automation of VLSI design [CW91, MPC90], the use of such systems is currently very limited in the industrial environments. Except for a few cases (such as CATHEDRAL [VRB+93], HIS [CBH+91], CALLAS [BNW92, BMK92]), high-level synthesis is not used in industry. The main problems are:

1 The lack of integration within existing CAD environments and design methodologies,

2 The lack of interaction with the designer and

3 Handling complex and heterogeneous design.

In order to be integrated within existing CAD environments, a behavioral silicon compiler must be compatible with simulation and other synthesis tools. Additionally, it must be able to interact with other similar synthesis tools. This implies to allow the use of blocks produced by other compilers as functional units and also to allow that the result of the behavioral silicon compiler itself can be used by other synthesis tools.

The arrival and acceptance of standard HDLs, namely, VHDL, Verilog, UDL/I etc. has provided a solution to the first problem. For example, AMICAL accepts a behavioral VHDL description as the input, and generates an RTL description of the circuit, also in VHDL. By using VHDL, AMICAL is compatible with the existing simulation environments at both levels of abstraction and thus allowing a functional verification. In addition, the output of AMICAL can interface with the existing logic synthesis tools by adapting the output of AMICAL to cater to different VHDL subsets offered by commercial logic-level synthesis tools.

For the second problem, much research still has to be carried out. In fact, a high-level synthesis process includes several NP-complete problems such as the scheduling, register allocation, FU allocation and so on. In addition, most of these steps are interrelated [MPC90, MPC88]. Although a lot of progress has been made towards finding

global and optimal solutions for the scheduling/allocation problem [PK89, GE92], heuristics and user-interaction are still necessary for the synthesis of large designs. A solution to this problem may be the use of interactive synthesis in order to allow mixing manual and automatic design within the high-level synthesis environment. This approach has already been experimented with several high level synthesis tools such as CATHEDRAL and AWB [TDW+88]. The solution provided by AMICAL consists of mixing manual and automatic design within an interactive, high-level synthesis environment. This approach will be detailed in the foregoing sections.

In order to handle complex designs, the synthesis process should allow the use of complex operators and functional units in order to hide complex behavior. Past experience with IC CAD tools has shown that hierarchy is a fundamental problem. The solution provided by AMICAL consists of using an external library of functional units. Each functional unit corresponds to an existing macro-block that may be used in the behavioral specification through procedure and function calls.

3.1.2 AMICAL: A global view

As shown in *Figure* 3.1, behavioral synthesis starts with two kinds of information; a behavioral description given in VHDL and an external library of functional units (FUs). The external library of FUs may include standard execution units such as the adders, multipliers, ALUs, etc. as well as more complex units defined by the designer. These may be large, complex blocks such as cache memories , I/O units, etc.

AMICAL is organized as an interactive environment wherein automatic and manual synthesis can be mixed. The response time of AMICAL is very short. Most of the examples synthesized by AMICAL required only a few seconds. Less than 10 minutes were required for the synthesis of a 300 Ktransistor complex industrial design, making it a genuine interactive system. The combination of automatic and manual synthesis allows a quick and broad exploration of the design space in real time. Furthermore, AMICAL provides many facilities for analyzing the generated architecture such as the statistics, evaluation, links between the VHDL behavior and the architecture, etc. The capabilities, strengths and weaknesses of AMICAL are detailed in the rest of this section. We will concentrate on the methodologies and the underlying concepts behind the development of AMICAL rather than on the individual algorithms. Details about the algorithms used by AMICAL can be found in [ORJ93].

.

REMERCIEMENTS

Cette thèse est le fruit d'un travail avec le laboratoire TIMA/INPG.

Je tiens à exprimer mes plus sincéres remerciements à:

Monsieur Bernard Courtois, directeur de recherche au C.N.R.S et directeur du laboratoire TIMA, pour m'avoir accueilli dans son laboratoire et pour m'avoir fait l'honneur de présider ce jury.

Monsieur Ahmed Jerraya, Charges de Recherche au C.N.R.S, et Responsable de l'équipe "System Level Synthesis Group" au TIMA, pour avoir dirigé mes recherches, pour sa participation à l'élaboration des idées et le temps qu'il m'a consacré pour cela.

Messieurs Eric Martin du LESTER, Professeur à l'université de Bretagne Sud et Jean-Pierre Schoellkopf Docteur d'Etat et Responsable de l'équipe Outils Avancés dans le Central de Recherche et de Developpement, SGS Thomson Microelectronics, Crolles pour m'avoir fait l'honneur dêtre rapporteurs de cette thèse, et qui ont accepté de faire le déplacement pour participer au jury de cette thèse.

Tous les (anciennes) membres de l'équipe de 'System Level Synthesis Group' à TIMA: Abderrazak Jemai, Inhag Park, Kevin O'Brien, Robert Lasoki, Mohamed Salmi, Khalil Kchouk, Mohamed Aichouchi, Bjarne Hald, Hong Ding, Tarek Ben Ismail, Polen Kission, Adel Changuel, Mohamed Abid, Mohamed BenMohammed, Elisabeth Berrebi, Wander Cesario, Jean-Marc Daveau, Louis DeGijsel, Julia Dushina, Jean Frehel, Philippe Guillaume, Fabiano Hessel, Christian Liabeuf, Clifford Liem, Gilberto Marchioro, Imed Moussa, Abdellatif Mtibaa, François Naçabal, Ahmed Ounaissa, Richard Pistorius, Maher Rahmouni, Mark Rietveld, Mohamed Romdhani, Rodolphe Suescun, Zoltan Sugar, Carlos Alberto Valderrama et Isabelle Es Salhiene.

Tous les membres du laboratoire et toutes les personnes qui ont contribué de près ou de loin à l'élaboration de ce travail.

Mes parents, ma soeur, mes frere et beau-frere qui ont eu la patience de me supporter dans la vie étrangère.

More Acknowledgements:

I'm indepted and wish to thank:

Dr. Philippe Garcin, Central R&D, SGS-Thomson Micro Electronics for showing me a progressive way in life and Dr. Schoellkopf for helping me to realize the same.

Mr. Olivier Ondoa, Researcher at the LGI laboratory, Grenoble, has been constantly and valuably helpful to me throughout my academic career in France.

All my DEA (French Master's) professors, the kind and helpful staff members of CIME, Dr.Pierre Gentil, the chairman of our Doctoral school.

Several of my well wishers, out of them are (alphabatically): H.Belhadj, E.Berrebi, A.Chagoya, V.Coissard, L. Costa, J.M.Daveau, H.Delori, R.O.Duarte, J.M.Karam, P.Kission, N.Krim, J.V.Medina, A.Mhani, M.Nicolaidis, A.B.Norena, R.Pistorius, M.Romdhani, S.Storojev, K.Torki, Isabelle, Corinne, Chantal, Patricia, ...

.

PREFACE

Le sujet traité dans cette thèse, concerne les liens entre la synthèse de haut niveau (HLS: High Level Synthesis) et la synthèse au niveau transfert de registres (RTL: Register Transfer Level). Il s'agit d'une adaptation de l'architecture résultat de la synthèse de haut niveau par transformation en une description (au niveau) RTL acceptée par les outils industriels actuels.

Les objectifs visés par cette transformation, sont: accroître la flexibilité et l'efficacité, permettre la paramétrisation de l'architecture finale.

A partir d'une description comportamentale décrite dans un language de description de materiel (la synthèse de haut niveau) génère une architecture au niveau transfert de registres, comprenant un contrôleur et un chemin de données. Le contrôleur et le chemin de données peuvent être synthétisés par des outils de synthèse RTL et logique existant pour réaliser un ASIC ou un FPGA. Cependant, pour des raisons d'efficacité, il est préférable de synthétiser le chemin de données par un compilateur de chemin de données.

Nous allons dans un premier temps concevoir une méthode que nous appelerons personnalisation. Elle permet aux concepteurs d'adapter l'architecture générée aux outils de synthèse RTL et à toute structure particulière requise.

Ensuite, nous définirons une méthode appelée Décomposition. Cette dernière fournira un moyen de décomposer un chemin de données en plusieurs sous chemins de données réguliers, pouvant être synthetisés de manière efficace par un compilateur de chemin de données.

Enfin, nous présenterons la génération de chemins de données génériques, destinés à la réalisation d'architectures paramétrables au niveau RTL. Cet algorithme a été implanté dans le generateur de code VHDL à partir de la structure de données intermédiaire utilisée par AMICAL, un outil de synthèse de haut niveau.

Le présent document est organisé en deux parties:

- *La première partie, en Francais, donne un apercu général des travaux accomplis, et des principaux résultats de la thèse.*
- *La seconde partie, en Anglais, est constituée de sept chapitres, et contient une description detaillée de l'ensemble des travaux et des résultats obtenus.*

We'll jointly develop technology that'll be able to put 10 million gates onto a single chip measuring about an inch square. Today's design tools can't handle this level of complexity.

– *Greg Ledenbach, Director of Design at SEMETACH Corp. June 3, 1996*

ENGLISH PREFACE

The microelectronics industry has been undergoing a pace of change in order to cope with the increasing complexity of VLSI. This thesis work addresses the subject of linking High Level Synthesis (HLS) of VLSI with the Register Transfer Level (RTL) synthesis prevailing to be the current Industrial practice. Starting from the HLS results, we generate quality RTL specifications equivalent of the initial behavioral specification. The generated results aim at an improved flexiblity, efficiency and parameterization from a designer's view point in terms of the final architecture.

A HLS starts from a behavioral description in a Hardware Description Language (HDL) such as the VHDL, performs a certain NP-complete steps such as scheduling, allocation to generate a RTL architecture based on a controller and a datapath. Both controller and datapath can be synthesized by the logic synthesis tools to realize an Application Specific Integrated Circuit (ASIC) or a Field Programmable Gate Array (FPGA). However, for efficieny reasons, it is preferable to synthesize the datapath using a datapath compiler. Further, the architecture obtained as a result of HLS can be parameterized.

We begin by developing a method called a Personalization to yield a flexible RTL architecture. This method allows the designers not only to add the information related to the synchronization, but also to mix in a high level description, both synthesizable and un synthesizable parts.

Next we define a method known as a Decomposition. It allows to transform an available HLS datapath into an interconnection of several regular datapaths and a glue logic. All the extracted regular datapaths can undergo efficient synthesis by a datapath compiler.

Finally, we present the delivery of generic datapaths supporting parameterizable architectures at the RT level. The idea is incorporated into a VHDL translator from the intermediate data structure used by AMICAL, a HLS tool.

The report is organized into two parts:

- *The first part, in French, gives a general summary of the accomplished work together with the principal results of the thesis work.*
- *The second part, in English, organized in seven chapters, contains detailed descriptions of the ensemble of work achieved and the results obtained.*

.

Contents

List of Figures

List of Tables

.

Part I

Présentation Générale des travaux de la thèse

.

0.1 INTRODUCTION

La synthèse de haut niveau (High Level Synthesis: HLS) ou synthèse comportementale des circuits VLSI est une méthodologie ou étape de conception assistée par ordinateur (Computer Aided Design: CAD). Elle génère une architecture au niveau transfert de registres (Register Transfer Level: RTL) à partir d'une spécification au niveau comportemental ou algorithmique. Les avantages de la synthèse de haut niveau sont les suivants [Syn94] :

1. Spécification et vérification rapides (à cause du volume réduit des spécifications),
2. Possibilité d'exploration de plusieurs architectures (car la synthèse est automatique),
3. Réduction du temps de simulation (il y a moins d'activités ou évènements),
4. Réutilisation facile des spécifications (permit grâce au haut niveau d'abstraction).

La principale caractéristique différenciant les niveaux d'abstractions comportamental et transfert de registres concerne leur concepts de temps. Au niveau comportemental, les opérations sont exprimées en terme de séquences d'étapes de contrôle. Par contre, au niveau transfert de registres, on utilise des cycles d'horloge pour fixer l'ordre d'exécution des opérations. Actuellement, les outils disponibles sur le marché permettent de synthétiser une architecture au niveau transfert de registres en ASIC aussi bien que sur un FPGA sous forme mer de cellules (Sea of Cells) ou mer de portes (Sea of gates) ou d'une matrice de portes logiques programmable (Field Programmable Gate Array: FPGA).

On peut décrire un système de synthèse de haut niveau au moyen des deux grandes étapes suivantes : la synthèse d'architecture et la génération d'architecture.

0.1.1 La synthèse d'architecture

La synthèse d'architecture convertit une spécification de haut niveau en une description au niveau transfert de registres. Elle se décompose principalement en les trois phases suivantes: la compilation de la spécification, l'ordonnancement, et l'allocation de ressources.

1. *La compilation*: la spécification initiale comportementale est un code source dans un langage de description de matériel (Hardware Description Language: HDL) tel que VHDL. Elle est compilée en une représentation interne, généralement un graphe de flot de données et/ou un graphe de flot de contrôle.

2. *L'ordonnancement* : L'ordonnancement permet de déterminer les étapes de contrôle ou états au cours desquels un nombre variable d'opérations sont exécutées. L'optimisation de la fonction de coût par l'ordonnancement requiert une combinaison du coût des ressources matérielles, du cycle d'exécution, et du nombre d'étapes de contrôle [CB90, Cam91, ORJ92, ORJ93, ROJ94, RJ95].

3. *L'allocation* : L'allocation des ressources comporte la sélection du type et de la quantité des modules matériels d'une bibliothèque, ainsi que la mise en correspondance de chaque opération avec le matériel sélectionné. Elle accomplit également les allocations de registres et des connexions.

0.1.2 Génération d'architecture au niveau transfert de registres

L'étape de génération d'architecture exploite le résultat de l'ordonnencament et de l'allocation pour générer une architecture composée d'une partie opérative et d'une partie contrôle. Elle constitue l'objet principal de cette thèse.

L'étape de génération d'architecture fournit une description qui peut être transmise à une synthèse de niveau inférieur. Cette description est composée d'une partie opérative et d'une partie contrôle. La partie opérative étant un assemblage de composants pris dans une bibliothèque. Elle peut aussi être mise à plat et synthétisée par ls outils de synthèse logique. Cette deuxième approche peut s'avérer coûteuse en termes de temps d'exécution CPU et de mémoire pour des grand circuits. Une seconde approche consiste à utiliser des outils spécifiques pour chaque partie du circuits. La partie contrôle peut être traitée par les outils de synthèse logique tandis que la partie opérative sera traitée par un compilateur de chemin de données.

0.1.3 Objectifs

L'objectif de ce travail de thèse est de concevoir, à partir d'une synthèse de haut niveau, une architecture au niveau transfert de registres qui soit flexible, efficace et paramétrable.

Une architecture est dite flexible, si elle permet au concepteur de la modifier de façon contrôlée. L'un des aspects importants que nous étudierons est la possibilité de combiner des descriptions synthétisables et non-synthétisables à un haut niveau. L'objectif est de réduire le volume de la spécification et d'utiliser des macro blocs déjà existants. Le problème de la flexibilité est traité par une méthode dite de "Personnalisation" expliquée dans la section 4.

Une architecture est considérée comme efficace si elle est en mesure de fournir de meilleurs résultats en terme de surface, de temps, de facilité de réutilisation, de modularité, etc. L'un des aspects majeurs que nous étudierons est la possibilité d'extraire

des chemins de données réguliers de l'architecture synthétisée à un haut niveau. Un compilateur de chemin de données permet d'obtenir une réalisation à haute densité, et qui soit paramétrable et modulaire. Ceci diffère des méthodes de synthèse utilisant un réseau booléen mis à plat. Nous proposons une méthodologie de décomposition détaillée dans la section 5 et dont la tâche est dans un premier temps d'extraire autant de chemins de données réguliers que possible, adaptés à un compilateur de données, puis dans un deuxième temps de reconstituer les résultats.

Une architecture est dite paramétrable, si sur sa totalité ou bien sur certaines de ses parties on peut appliquer différents paramètres. Les différents composants de l'architecture peuvent contenir des types de données fixes ou génériques. Le concepteur a la possibilité de modéliser les composants de l'architecture de façon fixe ou générique. Chaque composant générique a ses propres paramètres. Le problème de la paramétrisation est traité dans la section 6.

La section 2 présente l'état de l'art. Dans la section 3, nous détaillons le système de synthèse de haut niveau AMICAL [KDB+95, JPO93, Par92, CW91], qui constitue l'épine dorsale des méthodologies que nous détaillons dans les sections 4, 5 et 6. La section 7 clôt ce résumé et propose une "Introduction à la deuxième partie".

0.1.4 Contribution

La synthèse de haut niveau, utilise en général une bibliothèque de composants tels que les registres, multiplexeurs, additionneurs etc. Afin de favoriser une indépendance de la synthèse de haut niveau par rapport à la technologie, il est recommandé utiliser une bibliothèque de composants génériques. La plupart des outils de synthèse de haut niveau génèrent, un modèle au niveau transfert de registres utilisant un type d'architecture fixe, constitué d'un chemin de données irrégulier et d'un contrôleur. Pour des raisons d'efficacité, il est possible qu'il soit nécessaire de réorganiser ce modèle afin de mieux réaliser l'interface avec les outils de synthèse de plus bas niveau tels qu'un compilateur de chemin de données régulier. Les principales motivations de ce travail sont de pouvoir:

1. Combiner la spécification de haut niveau initiale avec des descriptions non synthétisables. La plupart des outils de synthèse de haut niveau n'admettent qu'un sous-ensemble restreint des langages de spécification. Ce sous-ensemble est combiné avec un style de codage particulier et considéré comme un sous-ensemble synthétisable. Cependant dans le cas de circuits réels complexes, les spécifications peuvent être mixtes et comporter des éléments synthétisables et des eléments non synthésisables. Ceci revient aux raisons suivantes: Les facteurs qui déterminent la présence

   (a) Macro-blocs réutilisables pour satisfaire aux exigences telles que le temps de commercialisation et/ou l'efficacité.

(b) Limitation due à des outils de synthèse de haut niveau (exemple: utilisation de multiple processus).

(c) Possibilité d'adaptation des différents aspects d'une architecture synthétisée tels qu'une synchronisation entre le contrôleur et le chemin de données.

(d) Exploration de l'espace de conception avec différents choix d'architectures.

2. Réaliser efficacement l'interface avec les outils de synthèse de plus bas niveau.

3. Gérer une bibliothèque de composants génériques.

Cette thèse peut étre considérée comme une suite des travaux de Aichouchi [Aic94]. Les travaux de ce dernier ont permis de definir le lien minimal entre AMICAL et les outils de synthèse niveau de transfert de registre. Ses travaux donnent une solution partielle au premier point cité plus haut, Ils [Aic94] ont permis de d'evelopper un lien entre AMICAL et la synthèse de bas niveau permettant d'insérer un modèle de synchronisation entre le contrôleur et la partie opérative. Les travaux de cette thèse donnent des solutions aux trois points cités ci-dessus.

Les principales contributions de cette thèse sont:

1. La définition d'une méthode appelée *personalisation* d'architecture permettant de combiner les parties synthétisables et les parties non synthétisables dans un contexte de synthèse de haut niveau.

2. L'introduction d'un concept de décomposition d'architecture permettant d'extraire les chemins de données réguliers contenus dans l'architecture synthétisée. Les chemins de données réguliers extraits peuvent être importés par un compilateur de chemin de données standard. Ces concepts ont été realisés et testés dans le cadre du système de synthèse de haut niveau AMICAL.

3. L'utilisation d'une bibliothèque de composants génériques pour générer la description RTL.

0.2 Etat de l'art

Bien que la littérature compte plus de 100 outils de synthèse de haut niveau [Mar79, Hil85, KNRR88, BCM+88, Wol91, BMK92, MSDP93], nous nous limiterons à la comparaison des 5 outils de synthèse de haut niveau suivants: *Cathedral* [SN91, RMVGG88] de l'IMEC, *HIS* [CBH+91] d'IBM , *VSS* de l'Université de Californie à Irvine, le compilateur comportemental (*Behavioral Compiler: BC*) de Synopsys et *AMICAL* du Laboratoire TIMA/INPG. Nous considèrerons tout particulièrement le problème de l'établissement d'un lien entre un outil de synthèse de haut niveau et les environnements existants de synthèse au niveau transfert de registres. Le *tableau* 0.1

résume les principales caractéristiques des outils de synthèse de haut niveau en considérant les différents aspects d'un lien entre la synthèse de haut niveau et la synthèse au niveau transfert de registres. Les caractéristiques de ces systèmes sont détaillées dans le chapitre 2 de la deuxième partie de cette thèse. Cette étude montre que parmi les outils étudiés seul AMICAL peut répondre aux quatre critères de flexibilité du *tableau* 0.1.

Problème	**Cathedral**	**HIS**	**VSS**	**Synopsys**	**Amical**
Différentes schémas de synchronisation	Non	Non	Non	Oui	Oui
Différentes bibliothèques	Oui	Non	Oui	Oui	Oui
Combiner les parties Synthétisables et non synthétisables	Non	Non	Non	Non/Oui	Oui
Différentes cibles d'outils de synthèse RTL	Oui/Non	Non	Oui	Non	Oui

Table 0.1: Comparaison des aspects des liens entre la synthèse de haut niveau et la synthèse RTL

Les avantages liés à l'utilisation d'AMICAL résident dans l'utilisation d'un langage intermédiaire efficace permettant d'apporter des modifications simples mais contrôlées à l'architecture synthétisée, et permettant en outre de:

1 supporter plusieurs schémas de synchronisations.

2 supporter plusieurs bibliothèques.

3 combiner les parties synthétisables et les parties non synthétisables.

4 s'adapter à plusieurs cibles pour la synthèse de bas niveau.

0.3 AMICAL

AMICAL [Par92, KDB+95, JPO93, CW91] est un système de synthèse de haut niveau. Partant d'une entrée VHDL, AMICAL génère une spécification complète pour les outils de synthèse existants au niveau logique et au niveau transfert de registres [OPJC93]. La synthèse comportementale utilisant AMICAL part de deux sortes d'informations: une description comportementale donnée en VHDL, et une bibliothèque externe d'unités fonctionnelles (UF). La bibliothèque externe d'unités

fonctionnelles peut comporter des unités d'exécution standards telles que des additionneurs, des multiplieurs etc., ainsi que des unités plus complexes définies par le concepteur. Ces dernières peuvent être: des blocs complexes telles que les mémoires caches, des unités Entrées/Sorties (E/S), etc. La sortie d'AMICAL est une architecture au niveau transfert de registres dans un format appelé SOLAR [JO92, PVD95]. SOLAR est un format à base de listes du style EDIF. La description SOLAR produite par AMICAL est constituée de deux parties: un contrôleur et un chemin de données. Une conversion de SOLAR à VHDL génère une description VHDL au niveau transfert de registres pour une synthèse logique utilisant les outils existants. L'architecture au niveau transfert de registres comprend aussi un contrôleur et un chemin de données.

Le contrôleur est une machine d'état fini de Mealy. Le chemin de données peut être une architecture à base de bus ou de multiplexeurs. Il est composé d'un ensemble de composants interconnectés entre eux. Les composants peuvent être typiquement des unités fonctionnelles, de unités de stockage, de multiplexeurs ou d'interrupteurs. Une architecture pipeline peut être aussi générée à la demande. Le pipeline prend en compte l'attente du chemin de données sur les signaux du contrôleur et vice-versa.

Une architecture pipeline a deux unités d'interface entre le contrôleur et le chemin de données. Une unité d'interface utilise un registre pour emmagasiner et libérer les signaux de manière synchronisée.

Une fois que les composants du chemin de données sont décrits ou disponibles au niveau transfert de registres, cette architecture est validée par simulation.

Cette simulation est comparée avec la simulation de la spécification initiale au niveau comportemental.

Le flot d'information du système AMICAL est illustré dans la *figure* 0.1.

0.4 Personnalisation

La personnalisation d'une architecture a pour objectif l'adaptation du résultat d'une synthèse de haut niveau en le rendant flexible pour la synthèse au niveau transfert de registres. La plupart des outils de synthèse de haut niveau cités dans la littérature semblent utiliser une architecture fixée au niveau transfert de registres.

0.4.1 Objectifs de la personnalisation

La méthode de personnalisation que nous proposons permet au concepteur d'inclure des détails techniques à l'architecture obtenue d'une synthèse de haut niveau. Bien que apparement simple, l'étape de génération d'architecture introduit quelques problèmes qui doivent être résolus afin de rendre la synthèse de haut niveau flexible et plus pratique. Considérons les aspects suivants :

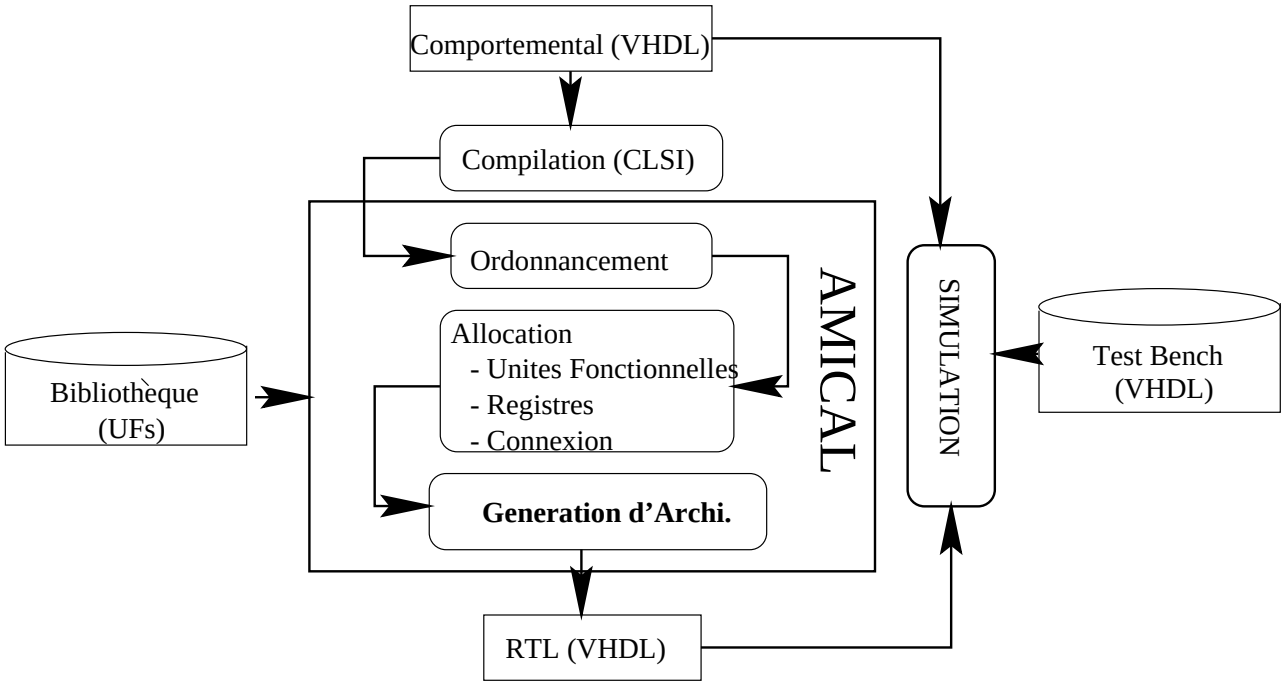


Figure 0.1: Flot d'information avec AMICAL

1. *Traiter plusieurs bibliothèques utilisant des conventions terminologiques différents.*

   L'architecture générée par une synthèse de haut niveau contient généralement plusieurs composants au niveau transfert de registres tels que des registres, des multiplexeurs et d'autres unités fonctionnelles. Au cours de la synthèse de haut niveau, nous utilisons des composants génériques qui doivent être mis en correspondance avec les composants existants. Les composants de la bibliothèque peuvent être désignés par des convention spécifiques. Par exemple, les termes désignant les signaux d'horloges et de réinitialisation (reset) ne sont pas standards ni identiques dans toutes les bibliothèques. Il est possible que nous ayons des bibliothèques des composants qui utilisent différentes terminologies pour ces signaux.

2. *Accepter différents schémas de synchronisation.*

   La synchronisation de haut niveau est abstraite en terme d'étape de contrôle et doit être mise en correspondance avec un schéma physique de synchronisation afin de réaliser une conception finale. Le résultat d'une synthèse de haut niveau est une architecture qui comporte un contrôleur et un chemin de données, et qui peut aussi admettre plusieurs schémas dédiés à la synchronisation de ces deux composants. Il nous faut donc adapter le schéma de synchronisation pour le niveau transfert de registres[1]. A cet effet, nous pouvons utiliser des signaux de liens externes pour le chemin de données, ou un bloc de distribution spécifique

[1] afin de réaliser l'architecture en fournissant des détails spécifiques au matériel.

d'horloge qui devra être rajouté dans le circuit.

3. *Combiner les descriptions synthétisables et non synthétisables*

   La description comportementale admise par les outils de synthèse de haut niveau constitue généralement un processus (unique) appartenant à la spécification d'un système plus grand. Après la synthèse de haut niveau, l'architecture résultante doit être reconnectée avec le reste du système englobant. De plus, de nombreux aspects concernant les interfaces (tels que les modules de commande de porte ou les broches de connexion : pads) sont difficiles à traiter au niveau comportemental.

Les trois problèmes mentionnés ci-dessus nécessitent une grande flexibilité de la part de l'étape de génération de synthèse d'architecture. Malheureusement, la plupart des outils de synthèse de haut niveau adoptent des solutions prédéfinies pour la plupart de ces problèmes.

0.4.2 Méthode

Notre méthode consiste, d'une part en la génération d'une architecture personnalisée à partir d'une architecture initiale disponible à partir d'une synthèse de haut niveau avec AMICAL, et d'un fichier de personnalisation (également appelé fichier global). Le fichier de personnalisation est un script dans un format à base de listes décrivant les transformations à accomplir sur l'architecture. Les transformations sont de type :

1. Addition de composants supplémentaires,
2. Addition de signaux,
3. Addition d'interconnexion locale.

L'étape de personnalisation rend plus flexible la génération de la description au niveau transfert de registres. Elle permet de combiner les descriptions synthétisables et non synthétisables dans les spécifications initiales de haut niveau (nécessaires à la validation) et d'obtenir ensuite de nombreux modèles de synchronisation au niveau transfert de registres. Le flot d'informations de la méthode de personnalisation est illustré par la *figure* 0.2.

La méthode de personnalisation est intégrée dans un outil appelé traducteur d'architecture programmable (Programmable Architecture Translator, PAT). Après avoir accompli une synthèse de haut niveau, le système AMICAL génère une architecture au niveau transfert de registres dans un format intermédiaire appelé Solar. On trouvera plus de détails sur le format Solar dans [O'B93]. La personnalisation est initiée sur le format Solar en utilisant un fichier de personnalisation et un ensemble de composants présents à l'intérieur d'une bibliothèque abstraite. On trouvera plus de détails sur le fichier de

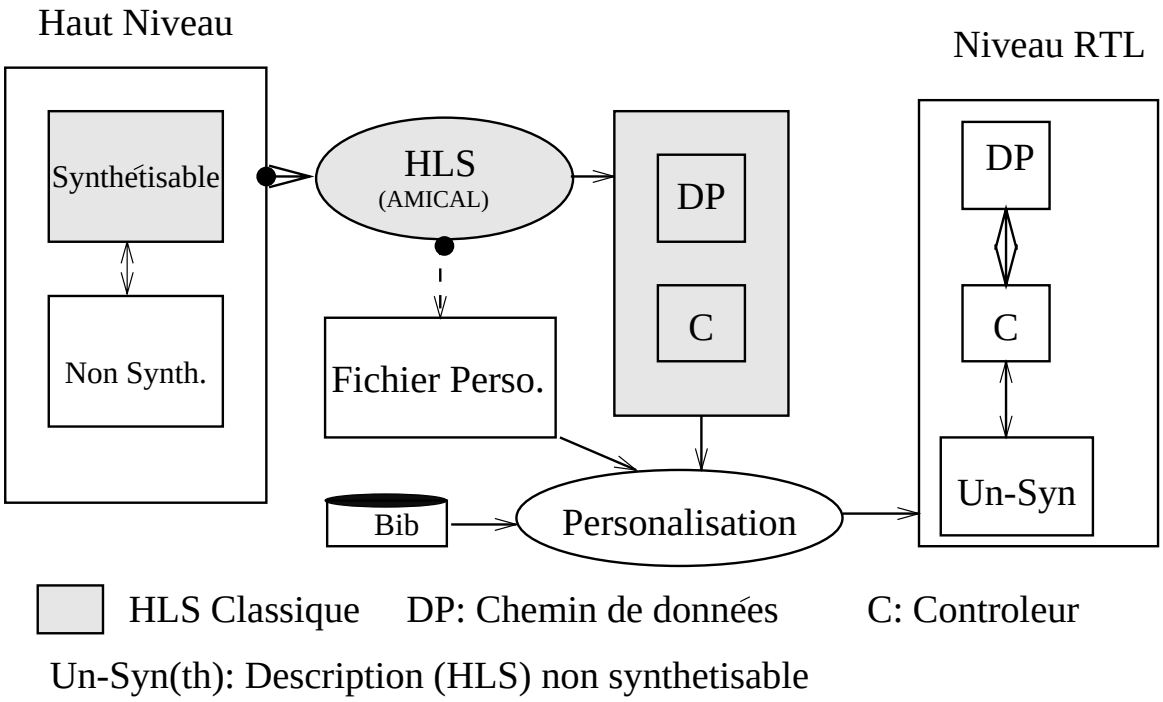


Figure 0.2: Flot d'information avec la méthode de personnalisation

personnalisation dans [Vij94b]. L'architecture, après personnalisation, contient des éléments du fichier de personnalisation ajoutés à l'architecture abstraite disponible à partir d'une synthèse de haut niveau. La personnalisation est l'un des facteurs clés du succès d'un outil de synthèse de haut niveau. Elle permet d'obtenir une architecture flexible au niveau transfert de registres. La personnalisation permet d'établir un lien entre le résultat de l'outil de synthèse de haut niveau et l'environnement existant au niveau transfert de registres.

0.4.3 Résultats

Nous considérerons deux exemples pour expliquer les résultats : Le premier exemple appelé "speed", est utilisé pour le contrôle de vitesse des moteurs de robots [CRJ96]. Le second exemple, appelé "motion", est un estimateur de mouvement utilisé avec un codec vidéo [BKV+96].

Le circuit de contrôle de vitesse est placé entre la mémoire et un ensemble de moteurs fonctionnant à des vitesses ("speed") différentes. La communication entre les deux côtés externes est établie au moyen des unités d'entrée sortie (E/S). Une unité d'entrée et de sortie (E/S) est une unité fonctionnelle spécifique qui peut échanger des données avec le monde extérieur sans intervention du contrôleur de vitesse. La *figure* 0.3 montre une partie du circuit de contrôle de vitesse réalisée en utilisant une synthèse de haut niveau et une personnalisation. Les deux unités d'interface appelées E/S Mem et E/S Moteurs sont utilisées dans le circuit de contrôle de vitesse comme

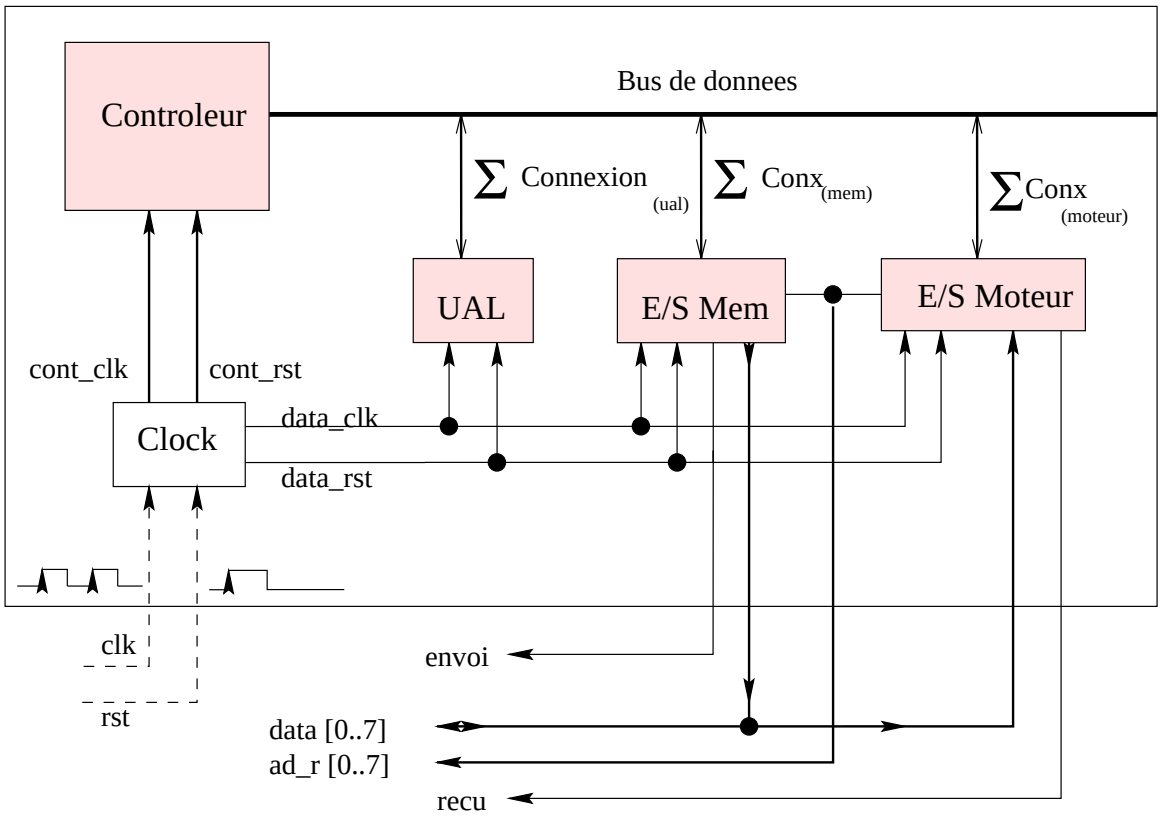


Figure 0.3: Partie du circuit de contrôle de vitesse

unités fonctionnelles simples permettant de lire et écrire dans les registres internes. Ces unités échangent des données avec le monde extérieur en utilisant un protocole sophistiqué basé sur les boucles "wait" et "handshaking". L'utilisation de ces unités d'interface évite la multiplication des procédures d'échange de protocole utilisées par le contrôleur principal. Le protocole est sous-traité par l'unité d'interface. Une partie du fichier de personnalisation utilisée par le circuit dans l'exemple "speed" est donnée ci-dessous :

ABBREVIATIONS:

s_u_l: `Std_Ulogic;` *s_u_l_v*: `Std_Ulogic_Vector.`

```
(global speed.global
   (block speed_circuit
      (signal clk       (dir In)       (attr Clock) (type s_u_l)  (local clk))
      (signal rst       (dir In)       (attr Reset) (type s_u_l)  (local rst))
      (signal cont_clk  (dir Internal) (attr Clock) (type s_u_l)  (local cont_clk))
      (signal cont_rst  (dir Internal) (attr Reset) (type s_u_l)  (local cont_rst))
      (signal data_clk  (dir Internal) (attr Clock) (type s_u_l)  (local data_clk))
      (signal data_rst  (dir Internal) (attr Reset) (type s_u_l)  (local data_clk))
      (signal send      (dir Out)      (type s_u_l)  (local SEND1))
      (signal recv      (dir Out)      (type s_u_l)  (local recv1))
```

```
        (signal (array data 8)        (dir InOut)  (type s_u_l_v)(local DATA1))
        (signal (array ad_r 8)        (dir Out)    (type s_u_l_v)(local Ad_R1))
        .
        .

        (glue    Clock)
    )
)
```

Dans la *Figure* 0.3, le résultat d'AMICAL correspond aux les blocs grisés et les lignes en gras. Les autres parties ont été rajoutées par l'étape de *personnalisation* en utilisant le script ci-dessus. Plus de détails sur cet exemple se trouvent dans la deuxième partie.

Le second exemple, l'estimateur de mouvement, utilise trois unités fonctionnelles : deux mémoires et un co-processeur DSP. La *figure* 0.4 montre une partie du circuit de l'estimateur de mouvement réalisé en utilisant une synthèse de haut niveau suivie d'une personnalisation. Dans le circuit ci-dessus, la communication entre les unités

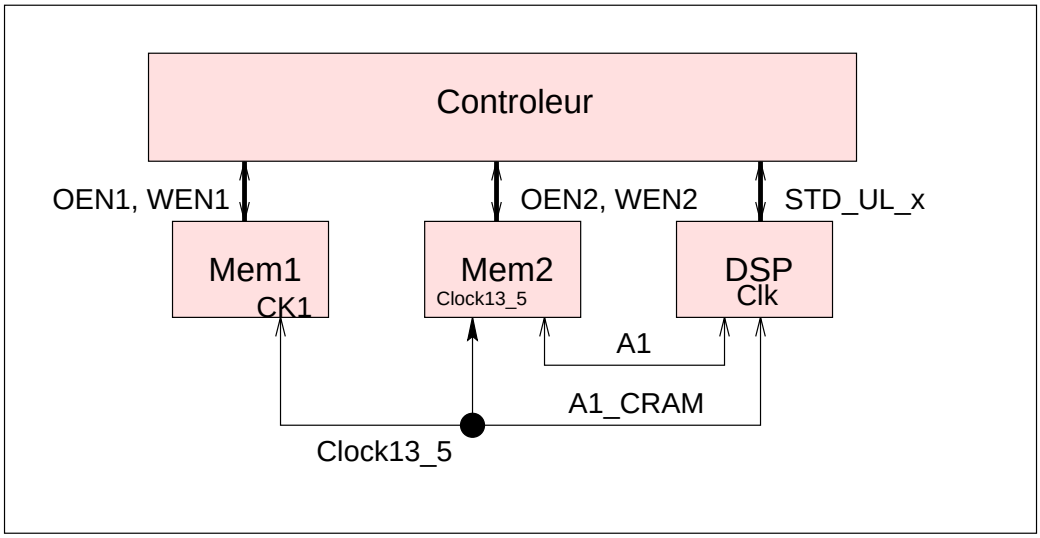


Figure 0.4: Partie de l'architecture du circuit de l'estimateur de mouvement

fonctionnelles et le monde extérieur est établie sans passer par le contrôleur principal.

1. Les exemples de communication entre les unités fonctionnelles sont illustrés par la *figure* 0.4 en utilisant les lignes A1 et le A1_CRAM. Ces exemples indiquent que le processeur utilise les mémoires sans que le contrôleur principal soit mis à contribution. En réalité, les unités fonctionnelles Mem1 et Mem2 sont utilisées comme des mémoires double port après une synthèse de haut niveau et une personnalisation. u.

2. Les exemples de communication entre unités fonctionnelles et le monde extérieur sont illustrés par la *figure* 0.4 en utilisant les ports CK1, clock13_5 et clk connectés par la ligne Clock13_5. Une fois de plus, une communication est possible sans mettre

à contribution le contrôleur principal. Cela permet d'obtenir une communication asynchrone avec le monde extérieur [BKV+96].

Une partie du fichier de personnalisation utilisée pour la personalisation du chemin de données de l'exemple de l'estimation de mouvement est donnée dans l'exemple ci-dessous. A l'aide de cet exemple, nous expliquons les différentes conventions terminologiques utilisées pour relier les composants existants.

Par exemple dans la première définition du signal : CK1, clock13_5, clk sont les ports de composants qui sont désignés par différents termes qui les connectent tous au signal dp_clk13_5.

```
(global motion.global
  (block motion_datapath
     (signal dp_clk13_5 (dir IN)
                        (attr clock)
                        (type std_ulogic)
                        (local CK1) (formal clock13_5) (formal clk))
     (signal (array std_ulv8_0 8)   (dir INTERNAL)
                        (attr CRAM)
                        (type std_ulogic_vector)
                        (local D2)  (formal std_ulv8_0))
     (signal std_ul_0   (dir INTERNAL)
                        (attr CRAM)
                        (type std_ulogic)
                        (local OEN2)(formal std_ul_0))
     (signal std_ul_1   (dir INTERNAL)
                        (attr CRAM)
                        (type std_ulogic)
                        (local OEN1)(formal WEN2) (formal std_ul_1))
     (signal (array A1_CRAM 8)      (dir INTERNAL)
                        (attr sel_CRAM)
                        (type std_ulogic_vector)
                        (local A1) (formal A1_CRAM))
     .
     .

  )
)
```

Le *tableau* 0.2 résume la liste des informations rajoutées lors de la personnalisation dans nos exemples servant à la démonstration. Le *tableau* 0.2 contient le nombre de signaux et de blocs rajoutés par la personnalisation.

Le support pour les conventions terminologiques (nommage de signaux à intercon-

Exemple	Contrôleur			Ch. de données			Circuit			Cellules residu	
	signal			signal			signal				
	Nom	Sync	Mix	Nom	Sync	Mix	Nom	Sync	Mix	Sync	Mix
SPEED	2	2	-	10	2	8	8	6	8	1	-
ME	-	4	5	13	3	26	1	11	15	1	1

Table 0.2: Signaux et cellules rattachés rajoutés via la personnalisation

necter) est fourni par la colonne nom, pour le schéma de synchronisation par la colonne Sync, et pour les spécifications mixtes synthétisables et non synthétisables par la colonne Mix.

Le mérite de l'exemple "speed" revient à A.Changuel et pour l'"estimateur de mouvement" à E.Berrebi, P.Kission et J.Frehel.

0.5 Décomposition

Rappel : une architecture résultant d'une synthèse de haut niveau comporte deux composants principaux : le contrôleur et le chemin de données.

0.5.1 Contexte

La méthode de décomposition d'architecture est appliquée au chemin de données de l'architecture au niveau transfert de registres. Le but est de restituer le chemin de données afin d'obtenir une synthèse efficace au niveau transfert de registres. Le chemin de données peut être synthétisé de deux manières différentes :

- Une synthèse logique du chemin de données permet une optimisation booléenne aboutissant à un réseau de portes logiques correspondant avec des cellules standards de la bibliothèque.
- Les composants au niveau transfert de registres peuvent être alternativement implémentés en utilisant un style d'implantation par tranche "bit-slice" afin de prendre en compte la nature régulière des composants du niveau transfert de registres. Cependant, si les composants ont une taille (nb. bits) différente, une grande partie de la surface d'implantation est gaspillée par le style bit-slice. Pour surmonter la déficience citée ci-dessus, nous proposons une technique de décomposition de chemin de données. Le but est d'extraire un ou plusieurs chemins de données réguliers (caractérisé par un nombre commune) de

l'architecture résultant de la synthèse de haut niveau (chemin de données) afin de le(s) soumettre à une synthèse du chemin de données. Si un certain nombre de composants tels qu'une ALU, un multiplexeur, un registre, un additionneur, ou un décaleur sont communs à différents chemins de données réguliers, alors ils peuvent être générés en utilisant un générateur paramétrable. Une fois que tous les chemins de données réguliers sont extraits, une logique de rattachement subsiste et peut être soumise à une synthèse logique (mise en correspondance de cellules standards). Il a été prouvé que la décomposition de chemins de données avait un impact significatif sur la surface finale d'implantation.

0.5.2 Méthode

La décomposition de chemins de données nécessite trois étapes successives :

1. Dans une première étape, appelée Flattening (mise à plat), chacune des instances présentes dans le chemin de données est étendue à son implémentation équivalente à un niveau plus bas, en termes de bibliothèques de composants de cellules standards et/ou de chemins de données. Ce procédé est similaire à l'approche de la synthèse fonctionnelle présentée dans [RG93b, AK94] .

2. L'étape suivante, appelée Splitting (partitionnement) a pour but l'extraction des chemins de données réguliers en partant d'une description du chemin de données disponible et mise à plat successivement. Au cours de chaque étape d'itération, le partitionnement exige un paramètre égal à la taille d'une chemin de données régulier pour être extrait. Après chaque extraction par partitionnement, la partie restante est assignée à un bloc de rattachement. Le processus se poursuit jusqu'à ce qu'il n'y ait plus de chemin de données régulier qui puisse être extrait du rattachement. Le bloc final de la logique de rattachement peut être synthétisé au niveau logique pour produire une implémentation optimale. Tous les chemins de données réguliers extraits seront étendus de manière bit-slice par le compilateur de chemin de données.

3. Au cours de l'étape finale, des traducteurs appropriés sont utilisés.

a Un chemin de données régulier est traduit dans un format EDIF pour être importé par le compilateur de chemin de données.

b La partie non régulière du chemin de données est traduite dans un format VHDL pour subir une synthèse logique.

c Le fichier "global" ne se traduit pas en VHDL. Cependent, les informations sont rajoutées lors de la traduction en VHDL pour être validé par simulation puis importé par les outils de synthèse de plus bas niveau vers le silicium.

Puisque les descriptions EDIF correspondent au réseau de chemins de données réguliers d'un bit, la taille des portes est spécifiée en utilisant des commentaires EDIF pour le concepteur. Les chemins de données réguliers sont étendus par le compilateur de chemin de données, placés et routés pour obtenir une implantation efficace. Les algorithmes de compression d'implantation tels que le routage de cellules, le pliage de piles etc., peuvent être appliqués pour obtenir une implantation de haute densité.

Un assemblage final des deux implantations (cellules standards et bit-slice) détermine l'implantation definitive du chemin de données. Le flot d'information pendant la décomposition du chemin de données est illustré par la *figure* 0.5.

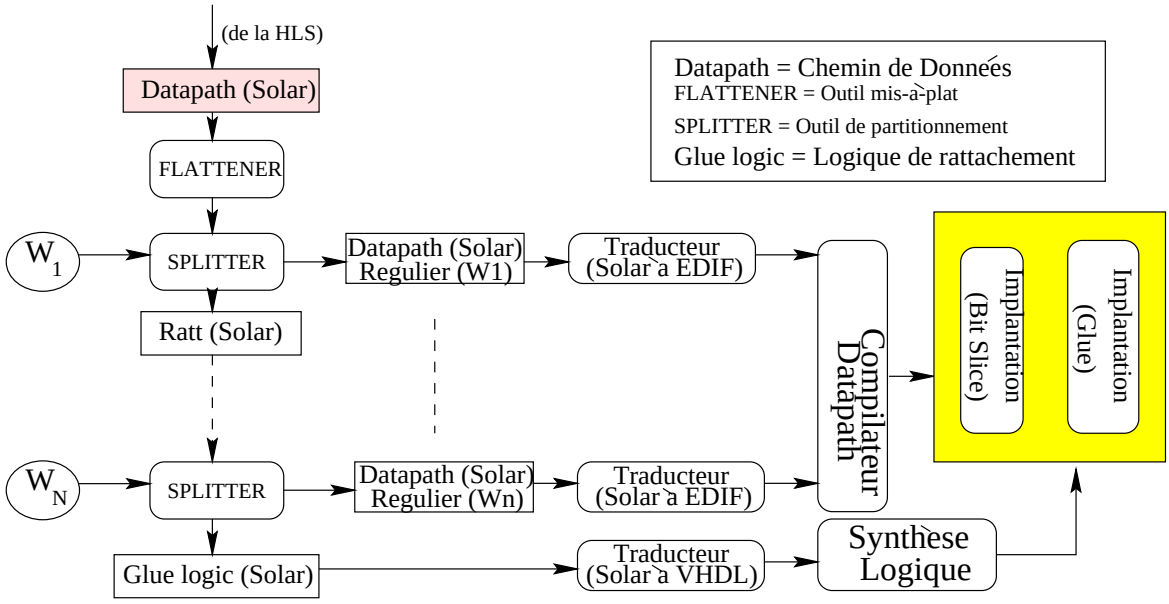


Figure 0.5: Flot d'information pendant une décomposition du chemin de données.

0.5.3 Surface

Après la décomposition, la surface des cellules du chemin de données peut être obtenue par addition des surfaces des parties opératives extraites de la surface, de la partie opérative residue de la décomposition. Nous donnons un exemple de chemin de données obtenu par une synthèse de haut niveau contenant deux chemins de données réguliers de taille 8 et 4 bits. Le chemin de données est décomposé en éléments de base d'une bibliothèque d'une technologie disponible. Un calcul théorique de la surface totale des cellules est présentée dans la *tableau* 0.3. Dans l'exemple ci-dessus, nous avons utilisé une technologie de 0,5 micron. Dans ce cas, nous réalisons une réduction de la taille du chemin de données de 0,426 mm2 à 0.143 mm2 en utilisant la méthode de décomposition avec un compilateur de chemin de données industriel.

Exemple $Surface_{Controleur}$ = 0.123 sq.mm (synthèse MEF) $Surface_{Datapath}$ = 0.426 sq.mm (synthèse logique)				
Bibliothèque	**Résultats de Décomposition**			**Surface**
	Datapaths Regulier			
Composants	nb. inst/bit	Surface (en termes de A_j)		in $sq.\mu$
(**Datapath**)		(W = 8)	(W = 4)	
Alu	$n_1 = 1$	$8.A_1$	$4.A_1$	13416
And-or	$n_2 = 4$	$32.A_2$	$16.A_2$	8448
Bus-stat	$n_3 = 2$	$16.A_3$	$8.A_3$	4224
Switch	$n_4 = 29$	$232.A_4$	$116.A_4$	91872
Inverter	$n_5 = 6$	$48.A_5$	$24.A_5$	6336
Ram	$n_6 = 1$	$8.A_6$	$4.A_6$	5280
Register-1	$n_7 = 1$	$8.A_7$	$4.A_7$	4224
Register-2	$n_8 = 2$	$16.A_8$	$8.A_8$	8448
	Total DP			142248
(**Logique Ratt.**)	nb.cells	Surface en A_j		
Inverter	$n_9 = 2$	$2.A_9$	-	220
Nand gate	$n_{10} = 4$	$4.A_{10}$	-	660
	Total GLUE			880
$Surf_{Datapath}$ après décomposition (cell area) = 0.143 sq.mm (after decomposition)				

Table 0.3: Résultats d'une décomposition d'un chemin de données

0.6 Paramétrisation

La paramétrisation d'architecture est une fonction du traducteur de l'architecture Solar en VHDL. Solar est la forme intermédiaire utilisée par AMICAL.

0.6.1 Objectif

La paramétrisation est utile pour réduire le nombre de déclaration de composants VHDL. Les instances présentes dans le chemin de données sont, soient des composants standards tels que les unités fonctionnelles, les unités de stockage, les interrupteurs, les unités de synchronisation, les unités de communication externe, les unités d'interface, etc., soient des composants complexes spécifiques des applications définies par l'utilisateur.

0.6.2 Méthode

Rappel : chacun des composants cités ci-dessus peut être de type générique ou non générique. Un composant générique contient au moins un paramètre générique, en plus de ses "ports" (E/S). Les paramètres génériques peuvent être utilisés pour paramétrer soit le composant lui-même (exemple : "ConstantValue" du "ConstReg") soit la taille de son "port", soit les deux. Le chemin de données peut être traduit en deux modes : générique et non générique. Le choix est laissé à l'utilisateur. Avec le choix du mode générique, il est possible d'utiliser certains des composants dans un mode non générique en ajoutant simplement une propriété "DATA_NON_Generic" à tous les "ports" de portType Data dans les composant(s) en format Solar de la bibliothèque des composants. Une telle utilisation est souhaitable pour la réutilisation d'un composant déjà conçu et de macro-blocs existants dont l'interface ne peut être modifiée par l'utilisateur (exemple: un DSP de 32-bits).

0.7 Conclusion

L'automatisation de la conception utilisant une synthèse de haut niveau donne aux concepteurs de systèmes électroniques plus de temps et de liberté lors des stades initiaux du cycle de conception. L'utilisation de la synthèse de haut niveau permet une approche descendante. L'objectif de notre travail de thèse est de concevoir une architecture au niveau transfert de registres qui soit flexible, efficace et paramétrable à partir d'une synthèse de haut niveau en vue de favoriser une réutilisation et une intégration. La flexibilité de l'architecture est réalisée par l'utilisation de la méthode appelée "personnalisation" et un fichier de personnalisation. Grâce à la personalisation les concepteurs peuvent combiner les parties synthétisables et non synthétisables des spécifications à un haut niveau. Ce qui permet, entre autre, une réutilisation de composants. L'efficacité de l'architecture est obtenue en séparant les parties régulières des parties opératives afin de les compiler à l'aide d'un compilateur de chemin de données régulier au lieu d'utiliser la synthèse logique. Ceci est réalisé par une méthode appelée décomposition qui consiste en un outil de mise à plat, un outil de partitionnement et un traducteur EDIF. Le résultat de la décomposition est un ensemble de chemins de données réguliers et un bloc résiduel. Ce dernier, après avoir été traduit en VHDL, est soumis à une synthèse logique en vue d'une optimisation booléenne. La paramétrisation d'architecture permet de paramétrer certains (ou tous) les composants du chemin de données afin qu'ils soient utilisés de manière générique plutôt que comme des cellules figées. La paramétrisation est réalisée par le traducteur VHDL. Le concepteur a la possibilité de générer des cellules génériques ou fixes dans la description VHDL. La méthode décrite ci-dessus a déjà été appliquée sur des exemples académiques aussi bien que sur des exemples industriels.

0.7.1 Introduction à la deuxième partie

Pour compléter cette section, un bref aperçu du contenu, de la partie suivante rédigée en anglais, est proposé. Chaque chapitre traite un problème spécifique et est relativement autonome.

Le Chapitre 1 situera le problème dans le contexte de la synthèse de haut niveau.

Le Chapitre 2 est une introduction à la synthèse de haut niveau et présente l'état de l'art.

Le Chapitre 3 sera dédié au système de synthèse de haut niveau AMICAL qui sera référé tout au long de ce travail de thèse.

Le Chapitre 4 décrit la méthode de personnalisation spécifique à l'architecture obtenue à partir d'AMICAL.

Le Chapitre 5 propose une méthode de décomposition du chemin de données pour extraire des chemins de données réguliers qui seront traités plus efficacement par un compilateur de chemin de données. La question de l'accès à la bibliothèque de composants sera également traité.

La Chapitre 6 regroupe les procédures de traductions de l'architecture dans les formats standards (VHDL et EDIF) au sein de l'environnement orienté objet.

Enfin, le Chapitre 7 conclura cette thèse en résumant la contribution apportée tout au long de ce travail à la méthodologie de la synthèse de haut niveau et en particulier au système AMICAL.

Part II

Exploration of the links between the High-Level Synthesis (HLS) and the Register Transfer Level (RTL) synthesis

.

Chapter 1

INTRODUCTION

The constant progress in the development of Integrated Circuits (ICs) is predicted to continue at least until the year 2010, reaching a complexity of 558 million transistors of a critical dimension 0.07 μ meters. Thanks to the High Level Synthesis (HLS), today, ICs may be designed starting from a specification at a high level (behavioural or algorithmic). HLS helps to explore the design space quickly and to convert the specification at the Register Transfer Level (RTL). An RTL specification consists of two parts: A controller *and a* datapath. *Controller is a Finite State Machine (FSM) specification for RTL controller synthesis. Datapath is a netlist of library components (functional, switching, ..) and can undergo either a logic synthesis for boolean optimization or a datapath synthesis for a parameterized functional expansion. This thesis work deals with the issue of linking behavioral synthesis with the lower level synthesis with an emphasis to current industrial practice. In this chapter, we shall position our problem in the above context.*

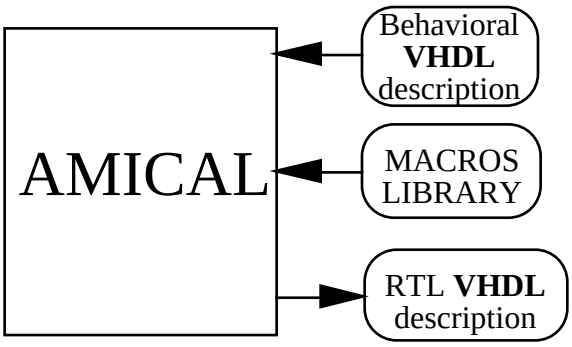


Figure 3.1: AMICAL: A global View

3.2 Integrating AMICAL with Existing Design Environments

3.2.1 Input description

The behavioral description model allows the handling of very large designs based on hierarchical specification. The basic idea behind this model is that a complex system is generally composed of a set of sub-systems performing specific tasks. A high-level specification of such a system needs only to describe the sequencing of these tasks, consequently the coordination of the different sub-systems. Each sub-system is modeled as a functional module designed (or selected) to perform a set of specific operations. Therefore the behavioral specification may be seen as a coordination of the activities of the different sub-systems. The decomposition of a system specification into a global control and detailed tasks allows the handling of very complex designs through hierarchy. The behavioral description is given as a standard VHDL file following a specific style. The use of complex sub-systems is made through procedure and function calls. For each procedure or function used, the library must include at least one functional unit able to execute the corresponding operation. During the different steps involved in the behavioral synthesis, the functional units are used as black boxes.

3.2.2 Using existing hardware

Functional units (FUs) are introduced in order to allow the mixing of behavior and structure, as well as the use of existing macro-blocks in the behavioral specification. An FU may execute a set of standard operations such as the adders, multipliers, ALUs, etc. or new, customized operations introduced by the user. It may be a large, complex block such as a cache memory, an I/O unit and so on. It can be called within a behavioral description in order to perform a given operation. It can accept and return parameters. In terms of hardware, it allows us to create partial designs

generate a complete RTL description. In this file, the designer defines the clock and reset signals, which are specific to lower levels of abstraction (RTL and downwards). However, (s)he can also specify additional blocks, that (s)he wants to connect to the generated architecture, as well as their connections. The application of the personalization rules on the different elements contained in a personalization file requires an understanding of one such a file [Pis96]. Appendix-6 contains the usage and grammar of a personalization file. The principal elements of this file are:

- Global_File_Contents ::= (**global** identifier_A *list_of_blocks*)

The starting keyword global defines a personalization file and is mandatory. This file can be empty if no personalization is required. The *identifier_A* is also mandatory and can be any string value passing the personalization file syntax parser. In general practice, this identifier is chosen to be significant to the personalization file. Then a list of blocks *list_of_blocks* will follow containing the personalization information required by the personalization of different circuit components.

- list_of_blocks ::= **empty** || (**block** identifier_U *block_contents*)

If there is any block (the circuit hierarchy) to be personalized, it must be specified here. The specification of such a block starts with the keyword block that is mandatory and preceeds the name of the part of the circuit *identifier_U*. The *identifier_U* must be identical to the name of the corresponding Solar DesignUnit. The *block_contents* are *signals* and/or *glue cells*. block *identifier_U*, corresponding to the SOLAR DesignUnit being treated, will be considered.

- block_contents ::= **empty** || *list_of_signals* &| *list_of_glue_cells*

If the specified block does not require any personalization, then the *block_contents* will be empty. If there is any signal in the *list_of_signals* and/or any cells in the *list_of_glue_cells*, the personalization of the specified block will be executed. The *list_of_signals* is a list containing zero or several signals of the following form:

- list_of_signals ::= **empty** || (**signal** *signal_name signal_qualifiers*)
- signal_name ::= identifier_V || (**array** identifier_V *signal_size*)
- signal_size ::= positive

Terminology

An object is a definition of type class. A node is instantiation of an object within the representation of a circuit. For example, the node *SolarNode* is an instantiation of the type SolarObject and a pointer to the SolarObject within the circuit is (*SolarObject *). All function names within the procedural interface terminate by a pair of open and close brackets "()". In the foregoing examples that either access or manipulate the database, the C++ terminology would be employed. The keywords in Solar, C++ and VHDL would be given in the *italics* wherever significant.

Primitive data types

The most primitive elements within the Solar intermediate format are used to represent the basic concepts of a circuit. It consists of structures of types arithmetic and boolean. There are 4 types of primitive data types:

data Type	Values
Boolval	(True, False)
Boolop	(nboolop, and, or, xor, eq, lt, gt, ge, le, ne)
Op	(nop, plus, minus, mult, div, mod, rshift, lshift)
Bitval	('0','1','X','Z')

The type **Bitval** contains four values corresponding to the logics for ground (0), power (1), unknown (X) and high impedance (Z). Infact, the **Bitval** is the basic subset of the Std_Logic type defined in the standard IEEE std_logic_1164 package.

There exists equally a data type primitive called **ObjectType**. It is an enumerated type containing a list of all the different nodes within the database.

ObjectType (SolarNode, PortNode, InstanceNode, ...)

Each element within the **ObjectType** corresponds to a structure of type class representing the different aspects of a Solar program. These classes are the objects of the database. A collection of attributes and operations permit the manipulation of the parameters assigned to each object. In the following section, we explain these objects, attributes and operations.

The Generic Types: Objects and Lists

The most fundamental data is called an *Object*. An *Object* is a structure of type class, containing a set of data types and methods to access them. The data types are eiter private (by default), public or protected. Two basic methods permit the construction (allocation of memory space, etc.) and destruction of the object (freeing the memory, etc.). All the attributes of an *Object* are automatically inherited by each node of its type created. The type of an attribute may be either a C++ type, or a Solar primitive

Appendix-3:
Typical VHDL Generic component

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity FlagReg is
    generic ( N : natural := 8);
    port(reset, clk: in STD_ULOGIC;
         R         : in STD_ULOGIC;
         W         : in STD_ULOGIC;
         Data_IN   : in STD_LOGIC_VECTOR (0 to N-1) := (others => 'Z');
         Data_OUT  : out STD_LOGIC_VECTOR (0 to N-1)  := (others => 'Z');
         CR        : out STD_ULOGIC_vector (0 to N-1)  );
end FlagReg;

architecture implementation of FlagReg is
   signal val_reg: STD_ULOGIC_vector(0 to N-1) ;
   signal L_W: STD_ULOGIC;
begin
   reg_file1: process
   begin
      wait until clk = '1' and clk'event;
      if reset='0' then
          val_reg <= (others => '0');
      elsif L_W = '1' then       -- Write --
          val_reg <= STD_ULOGIC_VECTOR(Data_IN);
      end if;
   end process reg_file1;
   reg_file2: process
   begin
         wait until clk = '1' and clk'event;
         if reset = '1' then
            L_W <= W;
         end if;
   end process reg_file2;
   CR <= val_reg;
   Data_OUT <= STD_LOGIC_VECTOR(val_reg);
end implementation;
```

.

Appendix-4:
Typical Flattened component in SOLAR

```
(SOLAR FU_FlagReg
(DESIGNUNIT FlagReg (view Implementation (viewType "Structure")
  (interface
   (port reset (direction IN))
   (port CLK (direction IN))
   (port R (direction IN))
   (port W (direction IN))
   (port Data_IN (direction IN))
   (port Data_OUT (direction OUT))
   (port CR (direction OUT)))
  (contents
   (instance D_1 (viewRef Implementation DPINV)
    (PortInstance Z (Property PortType DATA))
    (PortInstance A (Property PortType DATA)))
   (instance D_2 (viewRef Implementation DPA04)
    (PortInstance Z (Property PortType DATA))
    (PortInstance A (Property PortType DATA))
    (PortInstance B (Property PortType DATA))
    (PortInstance C (Property PortType DATA))
    (PortInstance D (Property PortType DATA)))
   (instance D_3 (viewRef Implementation DPA04)
    (PortInstance Z (Property PortType DATA))
    (PortInstance A (Property PortType DATA))
    (PortInstance B (Property PortType DATA))
    (PortInstance C (Property PortType DATA))
    (PortInstance D (Property PortType DATA)))
   (instance D_4 (viewRef Implementation DPREG)
    (PortInstance Q (Property PortType DATA))
    (PortInstance QN (Property PortType DATA))
    (PortInstance D (Property PortType DATA))
    (PortInstance C (Property PortType CONTROL))
    (PortInstance CN (Property PortType CONTROL)))
   (instance G_1 (viewRef Implementation IV)
    (PortInstance Z (Property PortType DATA))
    (PortInstance A (Property PortType DATA)))
   (instance G_2 (viewRef Implementation ND2)
    (PortInstance Z (Property PortType DATA))
    (PortInstance A (Property PortType DATA))
    (PortInstance B (Property PortType DATA)))
   (instance G_3 (viewRef Implementation ND2)
    (PortInstance Z (Property PortType DATA))
    (PortInstance A (Property PortType DATA))
```

```
    (PortInstance B (Property PortType DATA)))
   (net N_Data_IN (joined
          (portRef Data_IN)
          (portRef A (instanceRef D_1))))     ;; DPINV
   (net N_CR (joined
          (portRef CR)
          (portRef Z (instanceRef D_2))))     ;; DPA04
   (net N_Data_OUT (joined
          (portRef Data_OUT)
          (portRef Q (instanceRef D_4))))     ;; DPREG
   (net N_W (joined
          (portRef W)
          (portRef A (instanceRef G_1))       ;; DPINV
          (portRef A (instanceRef D_2))))     ;; DPA04
   (net N_RESET (joined
          (portRef RESET)
          (portRef A (instanceRef G_2))       ;; NAND2
          (portRef A (instanceRef G_3))))     ;; NAND2
   (net N_CLK (joined
          (portRef CLK)
          (portRef C (instanceRef D_4))))     ;; DPREG
   (net N_QN (joined
           (portRef QN (instanceRef D_4))     ;; DPINV
           (portRef B  (instanceRef D_2))     ;; DPA04
           (portRef B  (instanceRef D_3))))   ;; DPA04
   (net N1 (joined
           (portRef Z (instanceRef G_1))     ;; DPINV
           (portRef B (instanceRef G_2))     ;; NAND2
           (portRef D (instanceRef D_2))))   ;; DPA04
   (net N2 (joined
           (portRef Z (instanceRef G_2))     ;; NAND2
           (portRef B (instanceRef G_3))     ;; NAND2
           (portRef A (instanceRef D_3))))   ;; DPA04
   (net N3 (joined
           (portRef Z (instanceRef G_3))     ;; NAND2
           (portRef C (instanceRef D_3))))   ;; DPA04
   (net N4 (joined
           (portRef Z (instanceRef D_1))     ;; DPINV
           (portRef D (instanceRef D_3))     ;; DPA04
           (portRef C (instanceRef D_2))))   ;; DPA04
   (net N5 (joined
           (portRef Z (instanceRef D_3))     ;; DPA04
           (portRef D (instanceRef D_4))))   ;; DPREG
  )
 ))
)
```

Appendix-5:
Part of a Regular datapath in EDIF

```
(edif Amical_edif_for_DataPath  (edifVersion 2 0 0) (edifLevel  0)
  (keywordMap (keywordLevel 0))
  (status (written (timeStamp     1996 08 04 11 31 33);
      (program " Amical transEDIF " (version V 1.0 August 1996))
      (dataOrigin "Company") (author "designer")))
  (external DPHCMOS4T (edifLevel 0) (technology (numberDefinition))
      (cell DPINV (cellType GENERIC)
       (view Netlist_representation (viewType NETLIST)
         (interface   (port Z (direction OUTPUT))
                      (port A (direction INPUT)))))
      (cell DPRAM (cellType GENERIC)
       (view Netlist_representation (viewType NETLIST)
         (interface   (port clk (direction INPUT))
                      (port Sel (direction INPUT))
                      (port indata (direction INPUT))
                      (port inadr (direction INPUT))
                      (port outdata (direction OUTPUT)))))
      (cell DPALU (cellType GENERIC)
       (view Netlist_representation (viewType NETLIST)
         (interface   (port clk (direction INPUT))
                      (port Sel (direction INPUT))
                      (port in1 (direction INPUT))
                      (port in2 (direction INPUT))
                      (port out1 (direction OUTPUT)))))
      (cell DPBUSTAT (cellType GENERIC)
       (view Netlist_representation (viewType NETLIST)
         (interface   (port Z (direction OUTPUT)))))
      (cell DPA04 (cellType GENERIC)
       (view Netlist_representation (viewType NETLIST)
         (interface   (port Z (direction OUTPUT))
                      (port A (direction INPUT))
                      (port B (direction INPUT))
                      (port C (direction INPUT))
                      (port D (direction INPUT)))))
      (cell DPREG (cellType GENERIC)
       (view Netlist_representation (viewType NETLIST)
         (interface   (port Q (direction OUTPUT))
                      (port QN (direction OUTPUT))
                      (port D (direction INPUT))
                      (port C (direction INPUT)))))
      (cell IV (cellType GENERIC)
       (view Netlist_representation (viewType NETLIST)
         (interface   (port Z (direction OUTPUT))
```

```
                    (port A (direction INPUT)))))
    (cell ND2 (cellType GENERIC)
     (view Netlist_representation (viewType NETLIST)
       (interface   (port Z (direction OUTPUT))
                    (port A (direction INPUT))
                    (port B (direction INPUT)))))
    (cell DPENABTR (cellType GENERIC)
     (view Netlist_representation (viewType NETLIST)
       (interface   (port Z (direction OUTPUT))
                    (port D (direction INPUT))
                    (port E (direction INPUT))))))
(library DESIGNS (edifLevel 0) (technology (numberDefinition))
  (cell  bubble_mixed_datapath_DP_8  (cellType GENERIC)
    (view Netlist_representation(viewType NETLIST)
      (interface   (port  P_BUS_1_1 (direction  INOUT))
        (port  P_BUS_1_2 (direction  INOUT))
        (port  P_BUS_2_1 (direction  INOUT))
        (port  P_BUS_2_2 (direction  INOUT))
        (port  P_BUS_3_1 (direction  INOUT))
        (port  P_BUS_4_1 (direction  INOUT))
        (port  EBUS_1 (direction  OUTPUT))
        (comment "Port EBUS_1, Array Size is : 8")
        (port  EBUS_4 (direction  INPUT))
        (comment "Port EBUS_4, Array Size is : 8")
        (port  FLAG_t2 (direction  OUTPUT))
        (comment "Port FLAG_t2, Array Size is : 8")
        (port  FLAG_t3 (direction  OUTPUT))
        (comment "Port FLAG_t3, Array Size is : 8")
        (port  S_37_datain_BUS_1_1 (direction  INPUT))
        (port  S_36_ackin_BUS_3_1 (direction  INPUT))
        (port  S_35_ackin_BUS_2_1 (direction  INPUT))
        (port  S_34_validout_BUS_1_1 (direction  INPUT))
        (port  S_33_validout_BUS_3_1 (direction  INPUT))
        (port  S_32_dataout_BUS_1_1 (direction  INPUT))
        (port  S_2_0_BUS_3_1 (direction  INPUT))
        (port  S_1_BUS_1_1_0 (direction  INPUT))
        (port  CTRL_W_t3 (direction  INPUT))
        (port  S_3_1_BUS_2_1_t3 (direction  INPUT))
        (port  S_4_BUS_1_1_1 (direction  INPUT))
        (port  S_6_1_BUS_3_1 (direction  INPUT))
        (port  S_5_BUS_2_1_1 (direction  INPUT))
        (port  S_9_BUS_1_1_in1_FU_2 (direction  INPUT))
        (port  S_10_BUS_2_1_in1_FU_2 (direction  INPUT))
        (port  S_12_in2_FU_2_BUS_3_1 (direction  INPUT))
        (port  S_11_BUS_2_1_in2_FU_2 (direction  INPUT))
        (port  S_15_out1_FU_2_BUS_3_1 (direction  INPUT))
        (port  S_13_BUS_2_1_out1_FU_2 (direction  INPUT))
        (port  S_16_out1_FU_2_BUS_4_1 (direction  INPUT))
        (port  S_14_BUS_1_1_out1_FU_2 (direction  INPUT))
```

```
  (port  S_17_BUS_2_1_BUS_2_2 (direction  INPUT))
  (port  S_17_1_BUS_2_1_BUS_2_2 (direction  INPUT))
  (port  S_20_BUS_1_1_BUS_1_2 (direction  INPUT))
  (port  S_20_1_BUS_1_1_BUS_1_2 (direction  INPUT))
  (port  S_21_BUS_1_2_t2 (direction  INPUT))
  (port  S_22_BUS_2_2_t2 (direction  INPUT))
  (port  CTRL_W_t2 (direction  INPUT))
  (port  S_21_1_BUS_1_2_t2 (direction  INPUT))
  (port  S_22_1_BUS_2_2_t2 (direction  INPUT))
  (port  S_25_BUS_2_2_outdata_FU_1 (direction  INPUT))
  (port  P_N_167 (direction  INOUT))
  (port  P_N_165 (direction  INOUT))
  (port  P_N_163 (direction  INOUT))
  (port  P_N_161 (direction  INOUT))
  (port  P_N_135 (direction  INOUT)))
(contents
  (instance  dataout_U1
    (viewRef Netlist_representation (cellRef  DPINV)
                                    (libraryRef DPHCMOS4T)))
  (instance  FU_1_U1
    (viewRef Netlist_representation (cellRef  DPRAM)
                                    (libraryRef DPHCMOS4T)))
  (instance  FU_2_U1
    (viewRef Netlist_representation (cellRef  DPALU )
                                    (libraryRef DPHCMOS4T)))
  (instance  C1_U1
    (viewRef Netlist_representation (cellRef  DPBUSTAT )
                                    (libraryRef DPHCMOS4T)))
  (instance  C0_U1
    (viewRef Netlist_representation (cellRef  DPBUSTAT )
                                    (libraryRef DPHCMOS4T)))
  (instance  t2_D_2
    (viewRef Netlist_representation (cellRef  DPAO4 )
                                    (libraryRef DPHCMOS4T)))
  (instance  t2_D_4
    (viewRef Netlist_representation (cellRef  DPREG )
                                    (libraryRef DPHCMOS4T)))
  (instance  t2_G_1
    (viewRef Netlist_representation (cellRef  IV )
                                    (libraryRef DPHCMOS4T)))
  (instance  t2_G_2
    (viewRef Netlist_representation (cellRef  ND2 )
                                    (libraryRef DPHCMOS4T)))
  (instance  t3_D_2
    (viewRef Netlist_representation (cellRef  DPAO4 )
                                    (libraryRef DPHCMOS4T)))
  (instance  S_37_U1
    (viewRef Netlist_representation (cellRef  DPENABTR)
                                    (libraryRef DPHCMOS4T)))
```

```
            (net  datain_n3 (joined
              (portRef  A (instanceRef  datain_U2))
              (portRef  Z (instanceRef  datain_U1))))
            (net  t2_N_QN (joined
              (portRef  QN (instanceRef  t2_D_4))
              (portRef  B (instanceRef  t2_D_2))
              (portRef  B (instanceRef  t2_D_3))))
            (net  t2_N1 (joined
              (portRef  Z (instanceRef  t2_G_1))
              (portRef  B (instanceRef  t2_G_2))
              (portRef  D (instanceRef  t2_D_2))))
            (net  BUS_1_1 (joined
              (portRef  P_BUS_1_1)
              (portRef  D (instanceRef  S_20_U1))
              (portRef  Z (instanceRef  S_37_U1))
              (portRef  D (instanceRef  S_34_U1))
              (portRef  D (instanceRef  S_32_U1))
              (portRef  Z (instanceRef  S_14_U1))
              (portRef  D (instanceRef  S_9_U1))
              (portRef  D)
              (portRef  Z (instanceRef  S_4_U1))
              (portRef  Z (instanceRef  S_1_U1))
              (portRef  Z (instanceRef  S_20_1_U1))
              (portRef  Z)))
        )
    )
  )
)
(design Amical_edif_for_DataPath (cellRef bubble_mixed_datapath_DP_8
                                  (libraryRef DESIGNS)))
)
```

Appendix-6: Personalization

6.1 The Personalization file : Usage & Grammar

The personalization process from the AMICAL generated SOLAR files is enabled by a *personalization* or *synchronization* file (used to be called as *global* file) for architecture personalization. The personalization file is written in a list-based format such as *Lisp*, *EDIF* or *Solar* and is composed of:

- keywords
- a legal name for each keyword
- keyword qualifiers having legal values (if present)
- the rest of the contents, which in turn may contain another keyword

Every personalization file begins with the keyword *global*. Inline comments are allowed within the file, following a semi-colon character (;;). The contents of a personalization file may be (practically) empty; however, its presence is mandatory. [1] All keywords are case-insensitive (after VHDL). They may be written either using the upper or lower or mixed upper-lower case letters.

There are four principal keywords: **global**, **block**, **signal** and **glue**: *global* is the starting keyword of a global file; *block* is a part of *global* (any number (≥ 0) of *block* may be present within a *global*); *glue* is part of the *block* (any number (≥ 0) of *glue* may be present within a *block*); *signal* is part of the *block* (zero or any positive number of *signal* are allowed within each *block*) and has the following qualifiers: *dir*, *attr*, *type* and *local*.

The example for a **personalization** file shown below is the personalization file for the circuit description of the *Greatest Common Divisor* algorithm (GCD):

```
(global gcd.global
    (block gcd_control
        (signal cont_clock    (dir IN)
                              (attr clock)
                              (type Std_Ulogic)
                              (local clk))
        (signal cont_reset    (dir IN)
                              (attr reset)
                              (type Std_Ulogic)
                              (local reset)))
    (block gcd_datapath
        (signal data_clock    (dir IN)
                              (attr clock)
                              (type Std_Ulogic)
                              (local clk))
        (signal data_reset    (dir IN)
                              (attr reset)
                              (type Std_Ulogic)
                              (local reset)))
```

[1] A personalization file is said to be empty, when it contains no keyword other than *global*; such a file may be used if no personalization is desired.

```
(block gcd_circuit
    (signal clk          (dir IN)
                         (attr clock)
                         (type Std_Ulogic))
    (signal rst          (dir IN)
                         (attr reset)
                         (type Std_Ulogic))
    (signal cont_clock   (dir INTERNAL)
                         (attr clock)
                         (type Std_Ulogic)
                         (local cont_clk))
    (signal cont_reset   (dir INTERNAL)
                         (attr reset)
                         (type Std_Ulogic)
                         (local cont_reset))
    (signal data_clock   (dir INTERNAL)
                         (attr clock)
                         (type Std_Ulogic)
                         (local data_clk))
    (signal data_reset   (dir INTERNAL)
                         (attr reset)
                         (type Std_Ulogic)
                         (local data_reset))
    (glue Clock)
    )
)
```

Presented below are the keywords and qualifiers used by the personalization file of the gcd that is given above:

- **global** is the starting keyword of a personalization file and is used for the declaration of its name, here: gcd.global
- **block** declares the block to be personalized, here: use the personalization block **gcd_control** in order to personalize the file **gcd_control.solar** etc. **Remark:**
  - the block names, for example **gcd_datapath** (in this example) are case sensitive (UNIX) and shall not contain any of the shell meta characters
  - The files **gcd_control.solar**, **gcd_datapath.solar** and **gcd_circuit.solar** will be searched in the current directory
- **glue** represents an additional component to be added to the architecture, in the given example the cell *Clock* **Remark:**
  - **glue Clock** becomes a VHDL component instanciation after the translation into VHDL
  - *dir IN* corresponds to an input port
  - *dir OUT* corresponds to an output port

  - *dir INOUT* corresponds to an input-output port
  - *dir INTERNAL* corresponds to an internal connection (net)

- **signal** represents a signal definition within a block, in the given example of the gcd, there are two signals defined for the block *gcd_control*: **cont_clock** and **cont_reset**

- **attr** declares signal attributes, for example an attribute **Clock** has been declared for the signal *data_clock* within the block *gcd_datapath*

- **dir** defines the signal direction

- **type** valids the data type for the signal in the translated VHDL description. In the given example, the selected type is **Std_Ulogic**

- **local** maps a (VHDL) component's local port name to an actual signal name, in the given example of the gcd for the block *gcd_datapath*, the port name is **data_reset**

The syntax of the personalization file:

- Global_File_Contents
  ::= (**global** identifier_A *list_of_blocks*)

- list_of_blocks
  ::= **empty** ||
  (**block** identifier_U *block_contents*)

- block_contents
  ::= **empty** ||
  list_of_signals &|
  list_of_glue_cells

- list_of_signals
  ::= **empty** ||
  (**signal** *signal_name signal_qualifiers*)

- signal_name
  ::= identifier_V ||
  (**array** identifier_V *signal_size*)

- signal_size
  ::= positive

- signal_qualifiers
  ::= (**dir** *legal_value_for_DIR*)
  (**attr** *legal_value_for_ATTRIBUTE*)
  (**type** *identifier_D*)
  list_of_locals

- list_of_locals
  ::= **empty** ||
  (**local** *identifier_V*)
- list_of_glue_cells
  ::= **empty** ||
  (**glue** identifier_UV)
- legal_value_for_DIR
  ::= $IN||OUT||INOUT||INTERNAL$
- legal_value_for_ATTRIBUTE
  ::= $CLOCK||RESET||others$

Comments:

- Keywords are case insensitive: (*global* is the same as *Global* as well as *GLOBAL*.)
- Identifiers are case sensitive: (*gcd_datapath* is different from *Gcd_datapath*.)
- Definitions of the type *list_of_something* below mean that *something* may be repeated zero (empty) or more times.
- Definitions of the type *others* below mean that any other string value meeting the syntax would be accepted, but will have no semantic interpretation.
- The definitions *positive* below means any integer value greater than zero.
- *identifier_U* means any identifier string satisfying the UNIX file name conventions (with the exception of wild cards: $*, ?$).
- *identifier_V* means any identifier string that can be a legal name for a VHDL port/signal.
- *identifier_UV* means any identifier string that satisfies both of the above.
- *identifier_D* means any identifier string that can be a legal VHDL Data Type. (both Standard and Enumerated).
- *identifier_A* means any string value. (Must pass through the Global File Syntax parser)
- The symbol || stands for *OR*; && means *AND*; &| means *AND/OR*.

6.2 Options, Methods, Rules and Actions:

Personalization offers two different options for compiling and/or personalizing a solar description obtained after a High Level Synthesis using AMICAL:

```
-----------------------------------------------------------------------
Option                Purpose
-----------------------------------------------------------------------
CompileSolar          Syntax check & Data base creation
                      Input:  .solar file
                      Output: database , compile messages.

Personalize           Personalization of a solar file
                      Input:  .solar, .global files
                      Output: .solar (personalized) file.
-----------------------------------------------------------------------
```

Personalization is achieved using a set of methods (functions) in C++ and several rule based action tables. The methods are such as:

```
(char *) GetResolvedName()        : Ex: Std_logic_Vector
int      GetPortType()            : Data, Control, Unknown
int      GetWidth()               : >=1 if found, else 0.
int      GetComponentPortWidth()  : >=1 if found, else 0.
int      GetDirection()           : in, out, inout
int      GetConnectingNetMaxSize(): Maximum size of the connecting net.
int      IsGeneric()              : 0=No, 1=Yes if generic instance.
int      IsConnected()            : 0=No, 1=Yes if connecting interface.
int      IsInstancePort(IO *)     : 0=No, 1=Yes if an instance port.
int      IsGlobalPort             : 0=No, 1=Yes if port from global file.
int      GetGenericMapWidth_I(iO *) Find width from instance.solar file.
int      GetGenericMapWidth_G     : Find size from the global file
int      IsPortPresent (char *)   : 0=No, 1=Yes, if port name found.
```

Following sample action table contains a few rules and their corresponding actions. The rules given below are specific to the port instance objects (piO)s. The actions (a2 to a5A) are coded into the personalization program.

```
=======================================================================
Rule #  Variation      Conditions                    Value     Action
=======================================================================
2          -           piO->GetDirection() &&        OUT        a2
                       piO->GetPortKind()  &&        Vector
                       piO->IsPortConnection()       Yes
           ------------------------------------------------------------
           A           piO->IsPortConnKindBit()      Yes        a2A
           ------------------------------------------------------------
           A1          this-is_concurrent()          No         a2A1
           ------------------------------------------------------------
```

```
          B           piO->IsPortConnKindVector()  Yes        a2B
          ------------------------------------------------------------
          B1          this-is_concurrent()         No         a2B1
          ------------------------------------------------------------
          B2          this->IsResolved()  &&       No         a2B2
                      IsPortConnectionResolved()   No
          ------------------------------------------------------------
          B3.1        this-is_concurrent()         No         a2B31
          ------------------------------------------------------------
          B3.2a       this->IsResolved()  &&       Yes        a2B32a
                      IsPortConnectionResolved()   No
          ------------------------------------------------------------
          B3.2b       piO->IsResolved()   &&       Yes        a2B32b
                      IsPortConnectionResolved()   Yes
-----------------------------------------------------------------------
3         -           (piO->GetDirection() &&      IN         a3
                      piO->GetPortKind()  &&       Bit
                      piO->IsPortConnection() &&   No
                      piO->IsConnected() &&        Yes
                      (piO->GetConnNetBaseNameSz())
                      = piO->GetWidth()) ||        Yes
                      piO->GetPortType() ||        CONTROL
                      piO->GetPortType() ||        CLOCK
                      piO->GetPortType() ||        RESET
                      piO->GetPortType() ||        NONGENERIC
                      is_po_pc_file() ||           Yes
                      is_pc_po_file()              Yes
-----------------------------------------------------------------------
4         -           piO->GetDirection() &&       OUT        a4
                      piO->GetPortKind()  &&       Bit
                      piO->IsPortConnection() &&   No
                      piO->Is_alias           &&   No
                      piO->IsGlobalPort_generic()  No
-----------------------------------------------------------------------
5         -           piO->GetDirection() &&       IN         a5
                      piO->GetPortKind()  &&       Vector
                      piO->IsPortConnection()      Yes
          ------------------------------------------------------------
          A           piO->IsResolved() &&         No         a5A
                      piO->IsPortConnResolved()&&  Yes
                      piO->IsPortConnKindVec()     Yes
=======================================================================
```

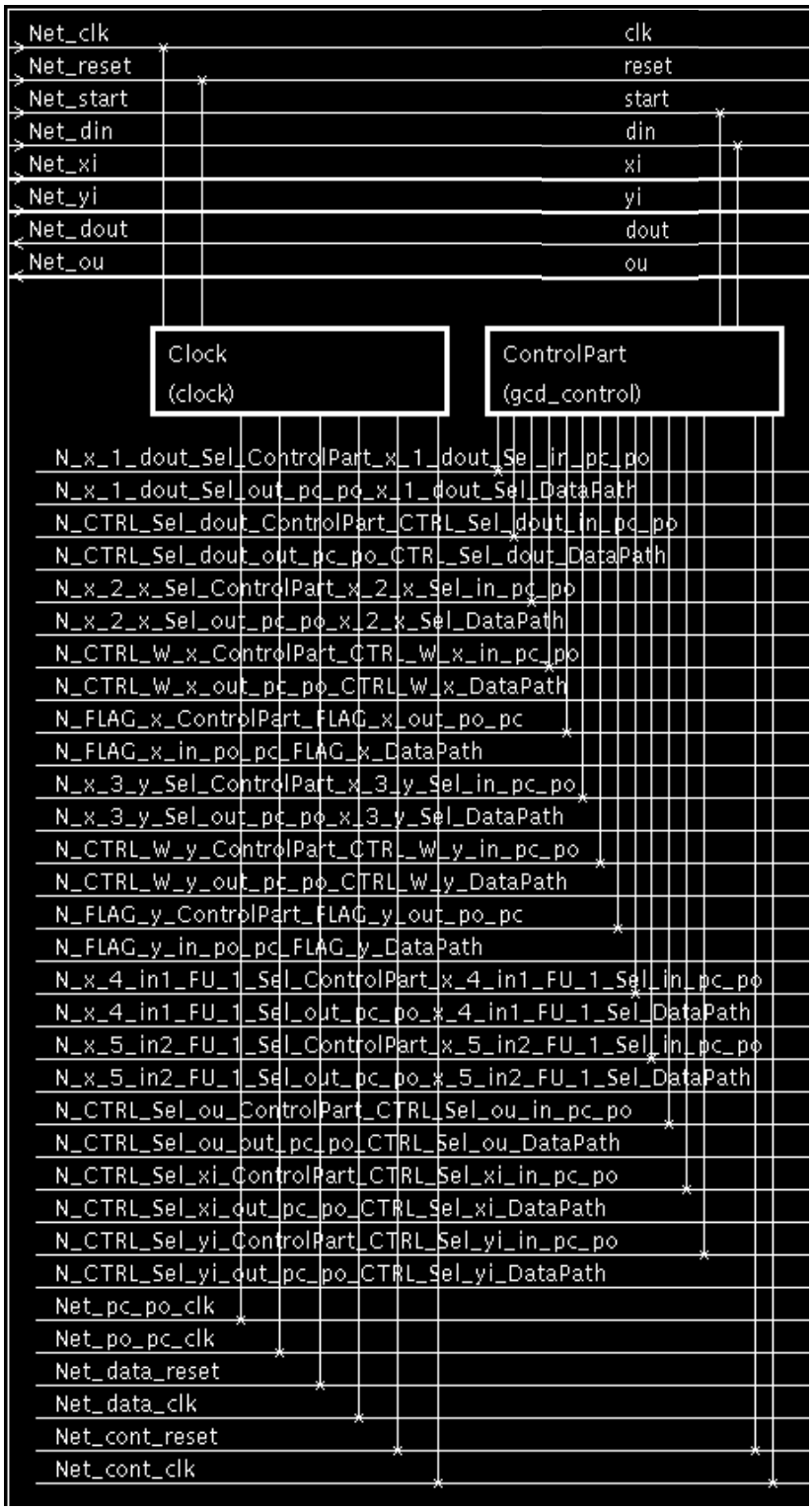


Figure .1: Effect of a *Glue* Personalization

6.3 Glossary:

Now, we present a glossary of some of the most useful functions realizing the personalization. The functions are classified according to the types of their return values:

short:

```
short IfaceObject::AreGenerics()
short InstanceObject::IsPortPresent(char *port_name)
short NetObject::DetWidth(char *WType)
short NetObject::GetBaseNameSize()
short NetObject::GetMaxWidth()
short NetObject::GetMinWidth()
short NetObject::IsMultiWidth()
short NetObject::IsPortConnection()
short PortObject::GetPortKind()
short PortinstObject::GetComponentPortType()
short PortinstObject::GetPortKind()
short PortinstObject::IsPortConnection()
short PortinstObject::IsPortConnectionKindBit()
short PortinstObject::IsPortConnectionKindVector()
short PortinstObject::IsPortConnectionResolved()
short PortinstObject::IsPortPresent2()
short PortrefObject::IsPortConnection()
short PortrefObject::IsPortConnectionKindBit()
short PortrefObject::IsPortConnectionKindVector()
short PortrefObject::IsPortConnectionResolved()
short SolarObject::IsDUPresent(char *du_name)
```

int:

```
int GetKind(ValueObject *Name)
int NetObject::IsMergable(NetObject *AnotherNet)
int PAT_SynchSignalObject::GetSignalSize()
int PAT_SynchSignalObject::GetSize()
int PAT_SynchSignalObject::IsArray()
int PAT_SynchSignalObject::IsFormal()
int PAT_SynchSignalObject::IsLocal()
int PortObject::GetDirection()
int PortObject::GetPortSize()
int PortObject::GetPortType()
int PortinstObject::GetComponentPortWidth()
int PortinstObject::GetConnectingNetBaseNameSize()
int PortinstObject::GetConnectingNetMaxSize()
int PortinstObject::GetDirection()
```

```
int PortinstObject::GetPortType()
int PortinstObject::GetWidth()
int PortinstObject::IsConnected()
int PortinstObject::IsDuplicate()
int PortinstObject::IsReallyGeneric()
int PortinstObject::IsResolved()
int PortrefObject::DetPortType()
int PortrefObject::GetWidth()
int PortrefObject::IsReallyGeneric()
int PortrefObject::IsResolved()
```

char pointers

```
char *InstanceObject::GetListOfPorts()
char *NetObject::DetName(char *Name)
char *NetObject::DetResolvedName()
char *NetObject::GetIrefName()
char *NetObject::GetPrefName()
char *NetObject::GetSignalbaseName()
char *PortObject::GetResolvedName()
char *PortinstObject::GetConcurrentStatementIn()
char *PortinstObject::GetConcurrentStatementInOut()
char *PortinstObject::GetConcurrentStatementOut()
char *PortinstObject::GetConnectingAliasNetBaseName()
char *PortinstObject::GetConnectingNetBaseName()
char *PortinstObject::GetConnectingNetName()
char *PortinstObject::GetResolvedName()
char *PortinstObject::GetUnconnectedSignalName()
char *PortinstObject::Getname()
char *PortrefObject::GetResolvedName()
```

.

Appendix-7: Datapath Decomposition

Datapath decomposition contains two steps: A hierarchy flattening and a splitting for the extraction of regular datapath(s).

7.1 Methods used in Flattening

Datapath flattening is a process to expand each of the instances present within a high level synthesized datapath into an equivalent structural network, the nodes of which representing the components of given technology libraries.

extern "C" void PrintFLATDataStruct()

```
{
  if (TopNode->IsA() == SolarNode) {
    SolarObject *SN = (SolarObject*) TopNode;
    if (SN != 0) SN->FLAT();
  }
  else printf("error %d\n", TopNode->IsA());
}
```

void SolarObject::FLAT()

```
{
  DUObject* du0 = ((DUObject*)this->designunit->head->e);
  if (designunit != 0) designunit->FLAT();
}
```

void DUObject::FLAT()

```
{
  if (name != 0) name->FLAT();
  if (property != 0) property->FLAT();
  if (view != 0) view->FLAT();
}
```

void ViewObject::FLAT()

```
{
  property->FLAT();
  if (interface != 0) interface->FLAT();
  if (procedure != 0) procedure->FLAT();
  if (contents != 0) contents->FLAT();
}
```

void IfaceObject::FLAT()

```
{
  operation->FLAT();
  property->FLAT();
```

```
  parameter->FLAT();
  port->FLAT();
  access->print();
  method->print();
}
```

void PortObject::FLAT()

```
{
  name->FLAT();
  direction->FLAT();
  if (typedval != 0)   typedval->FLAT();
  property->FLAT();
}
```

void ArrayObject::FLAT()

```
{
  name->FLAT();
  value->FLAT();
}
```

void ContentsObject::FLAT()

```
{
  variable->print();
  constant->print();
  statetable->print();
  instance->FLAT();
  net->FLAT();
  property->FLAT();
}
```

void InstanceObject::FLAT()

```
{
    int width_found = 0;
    List *pi = this->portinst;
    Link *pi_head, *pi_next;
    pi_head = pi_next = pi->head;

    if (pi_head != 0) {
     do {
     PortinstObject *pi0 = ((PortinstObject *) pi_next->e);
     Link *pr_head, *pr_next;
     pr_head = pr_next =  pi0->property->head;
     if (pr_head != 0) {
        do {
           PropertyObject *pr0 = ((PropertyObject *) pr_next->e);
           if (strcmp(strlower (SymbolTable[pr0->GetName()->GetIndex()]),
```

```
                              "width") == 0) {
            width = ((IntvalObject*)pr0->GetVal())->GetValue();
            width_found = 1;
               instance_visited = 1;
               break;
            }
            pr_next = pr_next->succ;
         }while (pr_head != pr_next);
        }
       if (1 == width_found) break;
   pi_next = pi_next -> succ;
   } while (pi_head != pi_next);
   }
  SolarObject *T = (SolarObject *) TopNode;
  yyparse();
  SolarObject *sn = (SolarObject *) TopNode;

DUObject* l = ((DUObject*)sn->designunit->head->e);
ViewObject     *v = l->GetView();
ContentsObject *c = v->GetContents();
List *inst = c->instance;
Link *inst_head, *inst_next;
inst_head = inst_next = inst->head;
if (inst_head != 0){
  do{
    InstanceObject* i = ((InstanceObject*)inst_next->e);
    i->GetViewref()->FLAT();
    i->portinst->FLAT();
    inst_next = inst_next->succ;
  } while (inst_head != inst_next);
}
List *net = c->net;
Link *net_head, *net_next;
net_head = net_next = net->head;
if (net_head != 0) {
   do {
     NetObject *n = ((NetObject *) net_next -> e);
     JoinedObject *jo = n->GetJoined();
     List *pref = jo->portref;
     Link *pref_head, *pref_next;
     pref_head = pref_next = pref->head;
     int internal_net = 1; // Internal Net ?
     if (pref_head != 0) {
        do {
        PortrefObject *pr = ((PortrefObject *)pref_next->e);
        InstrefObject *ir = pr->GetInstref();
        if (ir == 0) {
            internal_net = 0;  // NOT an Internal net.
            break;
```

```
            }
               pref_next = pref_next -> succ;
            } while (pref_head != pref_next);
         }
            char *internal_net_name;
            internal_net_name = (char *) malloc (30);
            strcpy (internal_net_name,
                 SymbolTable[n->GetName()->GetIndex()]);
         if (1 == internal_net) {
               pref_head = pref_next = pref->head;
               if (pref_head != 0) {
               do {
               PortrefObject *pr = ((PortrefObject *)pref_next->e);
               InstrefObject *ir = pr->GetInstref();
                     pref_next = pref_next -> succ;
               } while (pref_head != pref_next);
               }
         }
         net_next = net_next->succ;
     } while (net_head != net_next);
     }
}
```

void ViewrefObject::FLAT()

```
{
    name->FLAT();
    if (unitname != 0) unitname->FLAT();
    if (libname != 0)  libname->FLAT();
}
```

void NetObject::FLAT()

```
{
  name->FLAT();
  if (joined != 0) joined->FLAT();
}
```

void JoinedObject::FLAT()

```
{
  if (portref != 0) portref->FLAT();
}
```

void PortrefObject::FLAT()

```
{
 if (instref == 0) {
    if (value->IsA() == NameNode) {
       value->FLAT();
```

```
    }
}
else {
   char *port_ref_name;
   port_ref_name = (char *) malloc(100);
   strcpy (port_ref_name, SymbolTable[value->GetIndex()]);
   char *inst_ref_name;
   inst_ref_name = (char *) malloc(30);
   strcpy (inst_ref_name,
        SymbolTable[this->instref->GetName()->GetIndex()]);
   ContentsObject * c = (ContentsObject *)
               this->parent->parent->parent;
   List *inst = c->instance;
   Link *inst_head, *inst_next;
   inst_head = inst_next = inst->head;
   if (inst_head != 0) {
      char *inst_name;
      inst_name = (char *) malloc(100);
      do {
     InstanceObject *i = ((InstanceObject *)inst_next->e);
        strcpy (inst_name,
                  SymbolTable[i->GetName()->GetIndex()]);
        if (strcmp (inst_name, inst_ref_name) == 0) {
        FILE *fuin, *fuout;
        char FUFileName[30];
        strcpy(FUFileName,
               SymbolTable[i->GetViewref()->GetUnit()->
                           GetIndex()]);
        strcat(FUFileName, ".solar");
        fuin = fopen(FUFileName, "r");
        if (fuin == NULL) {
           cout << "\n        :-< File " << FUFileName
             << " is MISSING\t(required by PortrefObject::FLAT)\n";
         }
        else {
           SolarObject *T = (SolarObject *) TopNode;
           yyparse();
           SolarObject *sn = (SolarObject *) TopNode;
           DUObject *du = ((DUObject *)sn->designunit->head->e);
           ViewObject *vi = ((ViewObject *) du->view->head->e);
           ContentsObject *co = vi->GetContents();
           List *net = co->net;
           Link *net_head, *net_next;
           net_head = net_next = net->head;
              if (net_head != 0) {
             do {
                 NetObject *n = ((NetObject *) net_next->e);
                 JoinedObject *jo = n->GetJoined();
                 List *pref = jo->portref;
```

```
                Link *pref_head, *pref_next;
                pref_head = pref_next = pref->head;
                int net_visited = 0;
                if (pref_head != 0) do {
                     PortrefObject *pr =
                      ((PortrefObject *)pref_next->e);
                   char *leaf_port_ref_name;
                   leaf_port_ref_name = (char *) malloc(30);
                   strcpy (leaf_port_ref_name,
                           SymbolTable[pr->GetValue()->GetIndex()]);
                   if ( (0 == net_visited) && (pr -> instref == 0) &&
                        (strcmp (port_ref_name, leaf_port_ref_name) != 0) ) {
                      break;
                    }
                   else if ( (0 == net_visited) && (pr -> instref == 0) &&
                        (strcmp (port_ref_name, leaf_port_ref_name) == 0) ) {
                         pref_next = pref_next -> succ;
                         net_visited = 1;
                         continue;
                    }
                   if ((1 == net_visited ) && (pr -> instref != 0)) {
                             if (pr->value->IsA() == NameNode) {
                                  pr->value->FLAT();
                            }
                    }
                  pref_next = pref_next -> succ;
                 } while (pref_head != pref_next);
                net_next = net_next->succ;
            } while (net_head != net_next);
          }
             fclose (fuin);
          }
       }
       inst_next = inst_next->succ;
     } while (inst_head != inst_next);
     }
    delete port_ref_name;
    delete inst_ref_name;
  }
}
```

void PortinstObject::FLAT()

```
{
  if (name != 0) name->FLAT();
  if (property != 0) property->FLAT();
}
```

void NameObject::FLAT()

```
{
  fprintf(ttout," %s ", SymbolTable[index]);
}
```

void List::FLAT()

```
{
  Link *r, *f;
  r = f = head;
  if (r != 0){
    do{
      r->e->FLAT();
      r = r->succ;
    } while (r != f);
  }
}
```

void InstrefObject::FLAT()

```
{
  if (instref != 0) instref->FLAT();
  if (viewref != 0) viewref->FLAT();
}
```

7.2 Methods used by Splitting

A datapath splitting consists of identifying and extracting regular datapath(s) contained within a flattened solar description. Each regular datapath is characterized by a user given input parameter, usually called as a width.

extern "C" void PrintSPLITdpDataStruct()

```
{
  if (TopNode->IsA() == SolarNode) {
    SolarObject *SN = (SolarObject*) TopNode;
    if (SN != 0) SN->SPLITdp();
  }
  else printf("error %d\n", TopNode->IsA());
}
```

void SolarObject::SPLITdp()

```
{
  DUObject* du0 = ((DUObject*)this->designunit->head->e);
  if (designunit != 0) designunit->SPLITdp();
}
```

void DUObject::SPLITdp()

```
{
  if (property != 0) property->SPLITdp();
  if (view != 0) view->SPLITdp();
}
```

void ViewObject::SPLITdp()

```
{
  if (name != 0) name->SPLITdp();
  if (property != 0) property->SPLITdp();
  if (interface != 0) interface->SPLITdp();
  if (procedure != 0) procedure->SPLITdp();
  if (contents != 0) contents->SPLITdp();
}
```

void IfaceObject::SPLITdp()

```
{
  if (operation != 0) operation->SPLITdp();
  if (property != 0) property->SPLITdp();
  if (parameter != 0) parameter->SPLITdp();
  if (port != 0) port->SPLITdp();
  if (access != 0) access->print();
  if (method != 0) method->print();
}
```

void PortObject::SPLITdp()

```
{
  if (0 == PortObject_visited)  // single VISIT
  {
   if (0 == dpPort_splitting) dpPort_splitting = 1;
  /*********************************************************/
  /** Rule-2   : Datapath Port printing
  /** if (Port_width = dpSPLIT_WIDTH) && (PortType = DATA)
  /** Then, look for additional Port needs.
  /** Additional port needed whenever a DP instance connects
  /** to a Glue instance.
  /*********************************************************/
  char *dp_port_list;     // Global Variable is gl_dp_port_list, Vijay
  dp_port_list = (char *) malloc(num_port*port_name_size);
  char *dp_port_ref_list;
  dp_port_ref_list = (char *) malloc(num_port_ref*port_ref_name_size);
  ViewObject *vi0 = (ViewObject *)this->parent->parent;
  ContentsObject *co0 = ((ContentsObject *) vi0->GetContents());
  List *inst = co0->instance;
  Link *inst_head, *inst_next;
  inst_head = inst_next = inst->head;
  /*****************************************/
  /*** Instance Traversal                ***/
  /*****************************************/
  if (inst_head != 0) {
    int inst_found = 0;          // Is instance Found ?
    do {
      InstanceObject *in0 = ((InstanceObject *) inst_next->e);
      List *pi   = in0->portinst;
      Link *pi_head, *pi_next;
      pi_head = pi_next = pi->head;
      /*****************************************/
      /*** PortInstance Traversal            ***/
      /*****************************************/
      if (pi_head != 0) {
        do {
         PortinstObject *pi0 = ((PortinstObject *) pi_next->e);
         Link *pr_head, *pr_next;
         pr_head = pr_next =  pi0->property->head;
         /*****************************************/
         /*** PortInstance Property Traversal   ***/
         /*****************************************/
         if (pr_head != 0) {
           char   *porttype;
           porttype = (char *)  malloc(30);    // DATA or CONTROL
           int width = 0;
           do {
             PropertyObject *pr0 = ((PropertyObject *) pr_next->e);
             if (strcmp(strlower
```

```
                (SymbolTable[pr0->GetName()->GetIndex()]),
              "porttype") == 0) {
          strcpy(porttype, strlower
             (SymbolTable[pr0->GetVal()->GetIndex()]));
          if (strcmp(strlower(porttype), "control") == 0) {
            pr_next = pr_next->succ;
         continue;
        }
            }
            else if (strcmp(strlower
                       (SymbolTable[pr0->GetName()->GetIndex()]),
                     "width") == 0) {
            width = ((IntvalObject*)pr0->GetVal())->GetValue();
        }
         if ((strcmp (strlower(porttype), "data") == 0) &&
             (dpSPLIT_WIDTH == width )) {
         inst_found = 1;
               strcat(inst_list,
                   SymbolTable[in0->GetName()->GetIndex()]);
               strcat(inst_list,"$");
               break; }
           pr_next = pr_next->succ;
          }while (pr_head != pr_next);
        }
        if (1 == inst_found) break;
     pi_next = pi_next -> succ;
       } while (pi_head != pi_next);
   }
   inst_next = inst_next->succ;
   } while (inst_head != inst_next);
  }
List *net = co0->net;
Link *net_head, *net_next;
net_head = net_next = net->head;
/*****************************************/
/*** net Traversal                     ***/
/*****************************************/
if  (net_head != 0) {
  int net_found = 0;          // Is net Found ?
  do {
    NetObject *ne0 = ((NetObject *) net_next->e);
    JoinedObject *jo0 = ne0->GetJoined();
    List *pref = jo0->portref;
    Link *pref_head, *pref_next;
    pref_head = pref_next = pref->head;
    char *native_port_name;
    native_port_name = (char *) malloc(30);
    int native_port_found = 0;
    int instance_nature   = 0;
```

```
/*************************************/
/* Coding     Instance_nature        */
/*************************************/
/*    0       Initial value          */
/*    1       all DP instances       */
/*    2       all Glue instances     */
/*    3       Mixed DP & Glue instances */
/*************************************/
char *dp_port_ref_name, *dp_port_ref_name_dollar;
dp_port_ref_name = (char *) malloc(80);
dp_port_ref_name_dollar = (char *) malloc(80);
char *inst_string, *inst_string_dollar;
inst_string = (char *) malloc(inst_name_size);
inst_string_dollar = (char *) malloc(inst_name_size);
/*************************************/
/*** net PortRef Traversal          ***/
/*************************************/
if (pref_head != 0) {
  do {
 PortrefObject *pr0 = ((PortrefObject *)pref_next->e);
 if (((InstrefObject *)pr0 -> GetInstref()) == 0) {
   strcpy(native_port_name,
       SymbolTable[pr0->GetValue()->GetIndex()]);
      native_port_found = 1; }
    else {
   InstrefObject *ir0 = ((InstrefObject *) pr0->GetInstref());
      strcpy(inst_string,
        SymbolTable[ir0->GetName()->GetIndex()]);
      strcpy(inst_string_dollar, inst_string);
      strcat(inst_string_dollar, "$");
   if (strstr(inst_list, inst_string_dollar)) {
     if (0 == instance_nature) {
    strcpy (dp_port_ref_name,
         SymbolTable[pr0->GetValue()->GetIndex()]);
       strcpy (dp_port_ref_name_dollar,
         SymbolTable[pr0->GetValue()->GetIndex()]);
    strcat (dp_port_ref_name_dollar, "$");
    strcpy (dp_port_ref_list, dp_port_ref_name_dollar);
    instance_nature = 1; }
     else if (1 == instance_nature) {
    strcpy (dp_port_ref_name,
         SymbolTable[pr0->GetValue()->GetIndex()]);
       strcpy (dp_port_ref_name_dollar,
         SymbolTable[pr0->GetValue()->GetIndex()]);
       strcat (dp_port_ref_name_dollar, "$");
    strcat (dp_port_ref_list, dp_port_ref_name_dollar); }
        else {
       instance_nature = 3;                // Mixed inst.
    strcpy (dp_port_ref_name,
```

```
      SymbolTable[pr0->GetValue()->GetIndex()]);
    strcpy (dp_port_ref_name_dollar,
      SymbolTable[pr0->GetValue()->GetIndex()]);
    strcat (dp_port_ref_name_dollar, "$");
 strcat (dp_port_ref_list, dp_port_ref_name_dollar);
   if (!(strstr(dp_port_ref_list, dp_port_ref_name_dollar))) {
 strcpy (dp_port_ref_name,
      SymbolTable[pr0->GetValue()->GetIndex()]);
    strcpy (dp_port_ref_name_dollar,
      SymbolTable[pr0->GetValue()->GetIndex()]);
    strcat (dp_port_ref_name_dollar, "$");
    strcat (dp_port_ref_list, dp_port_ref_name_dollar);
    strcat (dp_port_ref_name, "_");
    strcpy (dp_port_ref_name,
         SymbolTable[pr0->GetValue()->GetIndex()]);
   } }
   IfaceObject *if0 = vi0->GetInterface();
   List *port = if0->port; // ask KEVIN
   Link *port_head, *port_next;
   port_head = port_next = port->head;
      /*****************************************/
      /*** Port Traversal                    ***/
      /*****************************************/
   if (port_head != 0) {
     do {
       PortObject *po0 = ((PortObject *) port_next->e);
          if ( !(strstr(dp_port_list, native_port_name)) &&
          (strcmp (strlower(native_port_name),
                   strlower(SymbolTable[po0->GetName()->
                            GetIndex()])) == 0)) {
         fprintf(dpout,"(port ");
         if (po0->name != 0)    po0->name->SPLITdp();
         if (po0->direction != 0)    po0->direction->SPLITdp();
         if (po0->typedval != 0)    po0->typedval->SPLITdp();
         if (po0->property != 0)    po0->property->SPLITdp();
         fprintf(dpout,")\n");
            // Global Port Names List, Vijay
            strcat(gl_dp_port_list, native_port_name);
            strcat(gl_dp_port_list, "$");

            strcpy(dp_port_list, native_port_name);
            strcat(dp_port_list, "$");
     }
     port_next = port_next -> succ;
      } while (port_head != port_next);
    }
 }
else {
  if ((0 == instance_nature) || (2 == instance_nature))
```

```
            instance_nature = 2;          // pure Glue
           else
            instance_nature = 3;          // Mixed
          }
           }
        pref_next = pref_next->succ;
      } while (pref_head != pref_next);
       }
       if ((0 == native_port_found ) && (3 == instance_nature)) {
              /*** Use P_<netname> for the Additional port. ****/
                 fprintf(dpout, "(port P_%s (direction INOUT) ",
                         SymbolTable[ne0->GetName()
                                     ->GetIndex()]);
                   // Global Port Names List, Vijay
                   strcat(gl_dp_port_list, "P_");
                   strcat(gl_dp_port_list,
                    SymbolTable[ne0->GetName()
                                ->GetIndex()]);
                   strcat(gl_dp_port_list, "$");

              fprintf (dpout, "(BIT) (property PortType DATA)");
                 fprintf(dpout,")\n");
       }
       net_next = net_next->succ;
     } while (net_head != net_next);
   }
   PortObject_visited = 1;
  }
}
```

void BitObject::SPLITdp()

```
{
  fprintf(dpout,"(BIT ");
  if (value != 0) value->SPLITdp();
  fprintf(dpout,")");
}
```

void BitvalObject::SPLITdp()

```
{
  switch(value) {
    case bit_zero : fprintf(dpout," '0'"); break;
    case bit_one  : fprintf(dpout," '1'"); break;
    case bit_x    : fprintf(dpout," 'X'"); break;
    case bit_z    : fprintf(dpout," 'Z'"); break;
  }
}
```

void ArrayObject::SPLITdp()

```
{
  fprintf(dpout,"(array ");
  if (name != 0) name->SPLITdp();
  if (value != 0) value->SPLITdp(); fprintf(dpout,")");
}
```

void DirObject::SPLITdp()

```
{
  fprintf(dpout,"(direction ");
  if (name != 0) name->SPLITdp(); fprintf(dpout,")");
}
```

void PropertyObject::SPLITdp()

```
{
  fprintf(dpout,"(property ");
  if (name != 0) name->SPLITdp(); //fprintf(yyout,"\n");
  if (valueref != 0) valueref->SPLITdp();
  if (strcmp(owner,"") != 0){
       fprintf(dpout,"\n");
       PrSPLITdpIndent();
     fprintf(dpout,"(Owner %s\")\n", owner);
  }
}
```

void ContentsObject::SPLITdp()

```
{
  fprintf(dpout,"(contents \n");
  if (instance != 0) instance->SPLITdp();
  if (net != 0) net->SPLITdp();
  if (property != 0) property->SPLITdp();
  fprintf(dpout,")\n");
}
```

void InstanceObject::SPLITdp()

```
{
   if (0 == dpInstance_splitting)
    {
     cout << "\n    <#> Phase 2-2 : DP Instance Splitting ...\n";
     dpInstance_splitting = 1;
    }
  /************************************************/
  /** Rule-1   : Datapath Instance printing
  /** Print ONLY
  /** if (PortType = DATA) && (Width = dpSPLIT_WIDTH)
  /************************************************/
```

```
int inst_found = 0;          // Is instance Found ?
List *pi   = this->portinst;
Link *pi_head, *pi_next;
pi_head = pi_next = pi->head;
if (pi_head != 0) {
   do {
   PortinstObject *piO = ((PortinstObject *) pi_next->e);
   Link *pr_head, *pr_next;
   pr_head = pr_next =  piO->property->head;
   if (pr_head != 0) {
      char   *porttype;
      porttype = (char *)  malloc(30);    // DATA or CONTROL
      int width = 0;
      do {
         PropertyObject *prO = ((PropertyObject *) pr_next->e);
            if (strcmp(strlower (SymbolTable[prO->GetName()->GetIndex()]),
         "porttype") == 0) {
         strcpy(porttype,
             strlower(SymbolTable[prO->GetVal()->GetIndex()]));
             if (strcmp(strlower(porttype), "control") == 0) {
                       pr_next = pr_next->succ;
                  continue;
               }
              }
            else if (strcmp(strlower (SymbolTable[prO->GetName()->GetIndex()]),
         "width") == 0) {
         width = ((IntvalObject*)prO->GetVal())->GetValue();
         }
         if ((strcmp (strlower(porttype), "data") == 0) &&
             (dpSPLIT_WIDTH == width ))
                { inst_found = 1;
                  break;
                   }
           pr_next = pr_next->succ;
          }while (pr_head != pr_next);
         }
      if (1 == inst_found) break;
   pi_next = pi_next -> succ;
   } while (pi_head != pi_next);
   }
if (1 == inst_found) {
fprintf(dpout,"(instance ");
if (name != 0) name->SPLITdp(); fprintf(dpout,"\n");
if (viewref != 0) viewref->SPLITdp();
if (portinst != 0) portinst->SPLITdp();
if (accinst != 0) accinst->SPLITdp();
if (property != 0) property->SPLITdp();
fprintf(dpout,")\n");
inst_found = 0;          // instance attended
```

```
  }
}
```

void ViewrefObject::SPLITdp()

```
{
    fprintf(dpout,"(viewRef ");
    if (name != 0) name->SPLITdp();
    if (unitname != 0) unitname->SPLITdp();
    if (libname != 0) libname->SPLITdp();
    fprintf(dpout,")\n");
}
```

void NetObject::SPLITdp()

```
{
  if (0 == dpNet_splitting) dpNet_splitting = 1;
  JoinedObject *jo0 = this -> GetJoined();
  List *pref = jo0 -> portref;
  Link *pref_head, *pref_next;
  pref_head = pref_next = pref->head;
  char *native_port_name = (char *) malloc (30);
  int native_port_found = 0;
  int native_port_found_save = 0;
  char *dp_port_ref_name, *dp_port_ref_name_dollar;
  char *dp_port_ref_name_save;
  dp_port_ref_name = (char *) malloc(80);
  dp_port_ref_name_save = (char *) malloc(80);
  dp_port_ref_name_dollar = (char *) malloc(80);
  char *inst_string, *inst_string_dollar;
  inst_string = (char *) malloc(inst_name_size);
  inst_string_dollar = (char *) malloc(inst_name_size);
  int net_printed = 0;
  int glue_connected = 0;     // Any connexion with Glue ?

  int dp_port_ref_name_size = 80;     // Max size of each port ref name;
      /*****************************************/
      /*** net PortRef Traversal             ***/
      /*****************************************/
      if (pref_head != 0) {
        do {
       PortrefObject *pr0 = ((PortrefObject *)pref_next->e);
          strcpy(dp_port_ref_name,
           SymbolTable[pr0->GetValue()->GetIndex()]);
       if (((InstrefObject *)pr0 -> GetInstref()) == 0) {
         strcpy(native_port_name,
             SymbolTable[pr0->GetValue()->GetIndex()]);
           native_port_found = 1;
           native_port_found_save = 1;
         }
```

```
 else {
InstrefObject *ir0 =
       ((InstrefObject *) pr0->GetInstref());
   strcpy(inst_string,
     SymbolTable[ir0->GetName()->GetIndex()]);
   strcpy(inst_string_dollar, inst_string);
   strcat(inst_string_dollar, "$");
if (strstr(inst_list, inst_string_dollar)) {// Part of DP
     if (1 == native_port_found ) {
       fprintf(dpout,"(net ");
       if (name != 0) name->SPLITdp();
       fprintf(dpout," (joined (portRef ");
       fprintf(dpout, native_port_name);
       fprintf(dpout,")\n");
       fprintf(dpout,"(portRef ");
       fprintf(dpout, dp_port_ref_name);
       fprintf(dpout," (instanceRef ");
       fprintf(dpout, inst_string);
       fprintf(dpout,"))  ");
 native_port_found = 0;
 net_printed = 1;
   }
     else {
 if (0 == net_printed) {
         fprintf(dpout,"(net ");
           if (name != 0) name->SPLITdp();
         char* P_net_name = (char *)
                malloc(net_name_size);
         strcpy(P_net_name, "P_");
         strcat(P_net_name, SymbolTable [this->GetName()
                        ->GetIndex()]);
         strcat(P_net_name, "$");
         fprintf(dpout," (joined ");
      if (  (strstr(gl_dp_port_list, P_net_name))) {
         fprintf(dpout," (portRef P_");
   if (name != 0)
        fprintf(dpout, "%s",
        SymbolTable[this->GetName()->GetIndex()]);
   else fprintf(dpout, "NoNET");
         fprintf(dpout,")");
         }
         fprintf(dpout," (portRef ");
   net_printed = 1;
  }
 else fprintf(dpout,"(portRef ");
         fprintf(dpout, dp_port_ref_name);
         strcpy(dp_port_ref_name_save, dp_port_ref_name);
         strcpy(dp_port_ref_name_dollar, dp_port_ref_name);
         strcat(dp_port_ref_name_dollar, "$");
```

```
                fprintf(dpout," (instanceRef ");
                fprintf(dpout, inst_string);
                fprintf(dpout,"))\n ");
            }
          }
          else
            if ( (1 == net_printed) &&
              (0 == native_port_found_save) ) {
              glue_connected = 1;
                 fprintf(dpout, " (portRef  ");
                 fprintf(dpout, dp_port_ref_name_save);
                 fprintf(dpout, ") ");
            }
         }
         pref_next = pref_next->succ;
        }while (pref_head != pref_next);
        }
        if (1 == net_printed) {
          fprintf(dpout,"))\n");
          net_printed = 0;
         }
}
```

void JoinedObject::SPLITdp()

```
{
  fprintf(dpout,"(JOINED \n");
  if (portref != 0) portref->SPLITdp();
  fprintf(dpout,")\n");
}
```

void PortrefObject::SPLITdp()

```
{
    fprintf(dpout,"(portRef ");
    if (value->IsA() == NameNode) value->SPLITdp();
     fprintf(dpout,"\n");
     if (instref != 0) instref->SPLITdp();
     fprintf(dpout,")\n");
}
```

void PortinstObject::SPLITdp()

```
{
  fprintf(dpout,"(portInstance ");
  if (name != 0) name->SPLITdp(); //fprintf(dpout,"\n");
  if (property != 0) property->SPLITdp();
  fprintf(dpout,")\n");
}
```

void NameObject::SPLITdp()

```
{
  fprintf(dpout," %s ", SymbolTable[index]);
}
```

void List::SPLITdp()

```
{
  Link *r, *f;
  r = f = head;
  if (r != 0){
    do{
      if (r->e != 0) r->e->SPLITdp();
      r = r->succ;
    } while (r != f);
  }
}
```

void InstrefObject::SPLITdp()

```
{
  fprintf(dpout,"(instanceRef ");
  fprintf (dpout, SymbolTable[this -> GetName() -> GetIndex()]);
  if (instref != 0) instref->SPLITdp();
  if (viewref != 0) viewref->SPLITdp();
  fprintf(dpout,")\n");
}
```

void IntvalObject::SPLITdp()

```
{
  fprintf(dpout," %d ", value);
}
```

7.3 Typical Datapath library Cell functions and their interfaces

```
NOTE: <> means a data port (N-bit wide, 1-bit in the input netlist)
      -- (due acknowledgements : Dr. Jean-Pierre Schoellkopf).


Buffer        : (A<>) (Z<>)
Inverters     : (A<>) (Z<>)
Logic AO1     : (A<>,B<>,C<>,D<>) (Z<>)
Logic AO2     : (A<>,B<>,C<>,D<>) (Z<>)
Logic AO2     : (A<>,B<>,C<>) (Z<>)
Ex-Nor        : (A<>,B<>) (Z<>)
Exor          : (A<>,B<>) (Z<>)
Nand2         : (A<>,B<>) (Z<>)
Nand3         : (A<>,B<>,C<>) (Z<>)
Nor2          : (A<>,B<>) (Z<>)
Nor3          : (A<>,B<>,C<>) (Z<>)
Or2           : (A<>,B<>) (Z<>)
Switch        : (D<>,E) (Z<>)
Bus Precharge : (CK) (Z<>)
Init-0        : (A<>) (Z)
Init-1        : (A<>) (Z)
Ground        : (A<>,FON) (Z<>)
Power         : (A<>,F1)  (Z<>)
And           : (A<>,B<>) (Z<>)
Decrement     : (D<>,CIN) (COUT,Q<>)
Increment     : (D<>,CIN) (COUT,Q<>)
Inc-Dec       : (D<>,CIN,INCDEC) (COUT,Q<>)
Comparator    : (A<>,B<>) (ANEB,AGEB)
Add-Sub       : (A<>,B<>,SUB,CIN) (COUT,S<>)
Add           : (A<>,B<>,CIN) (COUT,S<>)
Mult          : (X<>,Y<>) (PU<>,PD<>)
Multiplexor-2 : (D0<>,D1<>,S0,S0N) (Z<>)
Multiplexor-4 : (D0<>,D1<>,D2<>,D3<>,S0,S0N,S1,S1N) (Z<>)
Shifter-1     : (IS<>,ISC) (OSC,OS<>)
Shifter-2     : (IS<>,ISC0,ISC1) (OSC0,OSC1,OS<>)
Shifter-3     : (IS<>,ISC0,ISC1,ISC2) (OSC0,OSC1,OSC2,OS<>)
Shifter-4     : (IS<>,ISC0,ISC1,ISC2,ISC3) (OSC0,OSC1,OSC2,OSC3,OS<>)
Latch         : (D<>,C,CN) (Q<>,QN<>)
Latch-R       : (D<>,C,CN,RN) (Q<>,QN<>)
Latch-S       : (D<>,C,CN,SN) (Q<>,QN<>)
Latch-SR      : (D<>,C,CN,SN,RN) (Q<>,QN<>)
Register-R    : (D<>,C,CN,RN) (Q<>,QN<>)
Register-S    : (D<>,C,CN,SN) (Q<>,QN<>)
Register-SR   : (D<>,C,CN,SN,RN) (Q<>,QN<>)
Register JK   : (J<>,K<>,C,CN) (Q<>,QN<>)
Register JK-R : (J<>,K<>,C,CN,RN) (Q<>,QN<>)
Register JK-S : (J<>,K<>,C,CN,SN) (Q<>,QN<>)
Register JK-SR: (J<>,K<>,C,CN,SN,RN) (Q<>,QN<>)
```

Appendix-8: VHDL and EDIF Translations

The result of high level synthesis from AMICAL in the solar format can be translated into VHDL for Register Transfer Level or logic synthesis. On the other hand, the result of datapath decomposition in terms of regular datapaths can be translated into EDIF for import by regular datapath compilers.

8.0 AMICAL Programmable Architecture Translator (PAT)

```
Usage         : pat <command> <options>

  <command> can be any ONE of the following:
      -c      : Compile a Solar File      (and Quit)
      -f      : Solar to VHDL Translator (File Mode)
     -fg      : Solar to Generic VHDL Translator (File Mode)
      -q      : Quick execution grabbing more memory
      -h      : Help (That you've chosen :-)

  <options> may be ONE or MANY of the following (Position independent):
      -v   (*): Verbose
     -nv      : Suppress Verbose
      -w   (*): Print Warning Messages to a Log File called warnings.log
     -nw      : Suppress Warnings
      -i      : Treat the < next > argument as Input File name
      -o      : Treat the < next > argument as Output File name
      -g      : Treat the < next > argument as Global File name
      -ieee(*): Make reference to the IEEE std_logic_1164 package
      -mvl7   : Make reference to the Synopsys' MVL7 package
      -d2  (*): Use N downto 0 format for Port/Signal declarations
      -to     : Use 0 to N format for Port/Signal declarations
     Astericks(*)  above indicate Current Defaults

Related documents:

      1 PAT User's Manual
      2 How to write a Global file for PAT

Example Usage:

      solar -fg -nw -v -i gcd_datapath.solar
                       -o gcd_datapath.vhd
                       -g gcd.global
```

VHDL translation issues specific to the Controller (FSM):

1. A property called *arch_DP* indicates whether a generated architecture is mux (or) bus. If the architecture is a mux based one, then, the multiplexor select lines are initialized.
2. A *Synchronous process* is added to the controller description to make it function as a register transfer level finite state machine. Currently, the synchronous process uses a single clock sensitive to its rising edge and an asynchronous reset. However, it can be very easily extended to handling the choices such as:
   (a) synchronous versus asynchronous resets,
   (b) edge sensitivity to level sensitivity,
   (c) handling multiple clocks (phase, frequency, different edge sensitivities) etc.

VHDL translation issues specific to Datapath:

1. *Generic*: When a datapath description is translated using a generic mode, either some or all of the datapath components can be generic (VHDL) including the generics and the generic maps.
2. *Non Generic*: When a datapath description is translated using a non generic mode, all of the datapath components would be non generic (VHDL) and thus, not including any generics nor the generic maps.

Different options of Solar to VHDL translation

```
-----------------------------------------------------------------------
Option                Purpose
-----------------------------------------------------------------------

TRANSVHDL             Translation of solar to vhdl
                      Input:  .solar (personalized), <component>.solar
                      Output: .vhd

TRANSVHDLDESIGN       Translation of a design (many files) into vhdl.
                      Input:  .solar (personalized), <component>.solar
                      Output: .vhd for every .solar

TRANSVHDLGENERIC      Translation of a solar to generic vhdl file.
                      Input:  .solar (personalized) : Structural View.
                      Output: .vhd   (with generics & generic maps)
-----------------------------------------------------------------------
```

8.1 VHDL Translation methods

void Solar2VHDL_generic()

```
{
 if (TopNode->IsA()==SolarNode) {
     InitDES_generic();
     SolarObject *SN = (SolarObject*)TopNode;
     SN->vhdl_generic();
    }
}
```

Link *SolarObject::GetSynchLink()

```
{
 Link *TopLink, *TopHead;
 Link *SynchLink, *SynchHead;
 TopLink = TopHead = this->property->head;
 if (0 != TopLink) {
     do {
         if (TopLink->e->IsA()==SynchNode) {
             SynchLink = SynchHead =
                    ((SynchObject*)TopLink->e)->block->head;
             return (SynchLink);
        }
            TopLink = TopLink->succ;
         } while (TopLink != TopHead);
    }
 return 0;
}
```

void SolarObject::vhdl_generic()

```
{
 DUObject* du0 = ((DUObject*)this->designunit->head->e);
 path_AMICAL_PAT_libDIR = getenv ("AMICAL_PAT_libDIR");
 char *tv_duName = new char
         [strlen(SymbolTable[du0->GetName()->GetIndex()])+5];
 strcpy(tv_duName,SymbolTable[du0->GetName()->GetIndex()]);
 Link *GlobalSynchLink;
 GlobalSynchLink = this->GetSynchLink();
 GlobalBlockLink = this->GetBlockLink(GlobalSynchLink, tv_duName);
 if (strcasecmp ("Not_Available", PAT_InFileName))
    {
     ViewObject *v = du0->GetView();
     Link *PropLink, *PropHead;
     PropLink = PropHead = v->property->head;
     if (PropLink != 0) {
         do {
```

```
            PropertyObject *propO = ((PropertyObject *) PropLink->e);
            if (strcasecmp(SymbolTable
                [propO->GetName()->GetIndex()],"arch_dp") == 0) {
              char *OName = new char [strlen(SymbolTable
                [((PropertyObject*)PropLink->e)->GetVal()->
                  GetIndex()])+1];
              strcpy (OName, SymbolTable
                [((PropertyObject*)PropLink->e)->GetVal()->
                   GetIndex()]);
                if (0 == strcasecmp ("MUX",OName)) arch_DP = MUX;
                else if (0 == strcasecmp ("BUS", OName))
               arch_DP = BUS;
           delete OName;
           break;
               }
            PropLink = PropLink->succ;
           } while (PropLink != PropHead);
       } /*if PropLink != 0)*/
   }
 if (designunit->head != 0) designunit->vhdl_generic();
 delete tv_duName;
}

/*********************************************************************
* DUObject::vhdl_generic() requires:                                 *
*               DUBehaviour_generic(DUObject* NameObject*)           *
*               DUStructure_generic(DUObject* NameObject*)           *
* These functions are used to process designunits. If the ViewType   *
* is behavior the procedure DUBehaviour_generic is invoked; if it's  *
* structure, DUStructure_generic is called.                          *
*********************************************************************/
```

void DUObject::vhdl_generic()

```
{
  ViewObject *v = this->GetView (SymbolTable[this->GetName()->GetIndex()]);
  if (strcasecmp(v->GetViewtype(),"\"behavior\"")==0)
    DUBehaviour_generic(this, name);
  else if (strcasecmp(v->GetViewtype(),"\"structure\"")==0)
    DUStructure_generic(this, name);
  else if (strcasecmp(v->GetViewtype(),"\"communicating\"")==0){ }
  else if (strcasecmp(v->GetViewtype(),"\"portonly\"")==0){ }
  else if (strcasecmp(v->GetViewtype(),"\"port_only\"")==0){ }
  else if (strcasecmp(v->GetViewtype(),"\"component\"")==0){ }
  else{ DUBehaviour_generic(this, name); }
}
```

void DUBehaviour_generic(DUObject *DU, NameObject *name)

```
{
```

```
  GenDesUnit_generic(name);
  name->Pre_vhdl_generic();
  ViewObject *v = DU->GetView(SymbolTable[DU->GetName()->GetIndex()]);
  if (v != 0){
    IfaceObject  *iface = v->GetInterface();
    if (iface != 0) iface->Pre_vhdl_generic();
    name->Pre_vhdl_generic();
    end_of_entity = 1;
    v->Pre_vhdl_generic();
  }
}
```

void DUStructure_generic(DUObject *DU, NameObject *name)

```
{
  name->vhdl_generic();
  ViewObject *v = DU->GetView(SymbolTable[DU->GetName()->GetIndex()]);
  if (v != 0){
    IfaceObject  *iface = v->GetInterface();
    if (iface != 0) iface->vhdl_generic();
    name->vhdl_generic();
    end_of_entity = 1;
    v->vhdl_generic();
  }
}
```

void IfaceObject::Pre_vhdl_generic()

```
{
 Port_Personalize(parent,port);
 AddSynchPorts_generic(parent,port);
 port->Pre_vhdl_generic();
}
```

void IfaceObject::vhdl_generic()

```
{
  AddSynchPorts_generic(parent,port);
  Port_Personalize(parent,port);
  IsEntityPort = ONE;
  port->vhdl_generic();
  IsEntityPort = ZERO;
}
```

void PortObject::vhdl_generic()

```
{
  if(name->IsA()==NameNode){                    // Bit Node
    name->vhdl_generic();
    direction->vhdl_generic();
```

```
    if(typedval!=0) typedval->vhdl_generic();
    PrintVirg_generic(name,";");
  }
}
```

void DirObject::vhdl_generic()

```
{
  name->vhdl_generic();
}
```

void ViewObject::vhdl_generic()

```
{
  if (contents != 0) contents->vhdl_generic();
}
```

void ContentsObject::vhdl_generic()

```
{
  ViewObject *vw=(ViewObject*) parent;
  vw->GetName()->vhdl_generic();
  DUObject *du = (DUObject*) vw->parent;
  du->GetName()->vhdl_generic();
  DeclSynchSignals_generic(parent);
  if (variable != 0) variable->vhdl_generic();
  if (constant != 0) constant->vhdl_generic();
  net->Signaldecl_generic();
  this->DeclOpenInSignals();
  if (0 != vw) IfaceObject  *iface = vw->GetInterface();
  DeclSynchComponents_generic(parent);
  instance->vhdl_generic();
  DeclSynchPortMap_generic(parent);
  instance->Setportmap_generic();
  vw->GetName()->vhdl_generic();
}
```

void ContentsObject::DeclSynchComponents_generic(Object* p)

```
{
  Link *TopLink,   *TopHead;
  Link *BlockLink, *BlockHead;
  Link *SigLink,   *SigHead;
  Link *GlueLink,  *GlueHead;
  Link *PropLink,  *PropHead;
  char *BlockName = GetUnitName_generic(p);
  ViewrefObject *vr = ((InstanceObject *) this->instance)->GetViewref();
  TopLink = TopHead = GlobalSN->property->head;
  if ((0 != TopLink) && (TopLinkIndex != TopLink)) { TopLinkIndex = TopLink; }
  if  (TopLink != 0){
```

```
      /* Declare Synch Components */
  } /*if (Toplink != 0...*/
}
```

void ContentsObject::DeclSynchPortMap_generic(Object* p)

```
{ /* Declare Synch Components' Port's map */
}
```

void ConstObject::vhdl_generic()

```
{
  value->vhdl_generic();
  if (typedval != 0) typedval->vhdl_generic();
}
```

void VarObject::vhdl_generic()

```
{
  if (value->IsA() != ArrayNode)
    fprintf(ttout,"   signal ");
  value->vhdl_generic();
  if (typedval != 0)
    typedval->vhdl_generic();
  fprintf(ttout," ;\n");
}
```

void ContentsObject::DeclOpenInSignals()

```
{ /* Declare Open Input Signals */
}
```

void NetObject::Signaldecl_generic()

```
{ /*Declaration of all architecture signals*/
}
```

void InstanceObject::vhdl_generic()

```
{ /*Declaration of all components */
}
```

void InstanceObject::Setportmap_generic()

```
{ /* The Synchronisation components' port maps were declared in
  DefSynchPortMap_generic(). The other instantiations are
  treated here */
}
```

int PortObject::IsResolved()

```
{ /* Check if port is of type Resolved */
  int returnval = 0;
  Link *PropLink, *PropHead;
  PropLink = PropHead = this->property->head;
  if (PropLink != 0){
    do{
       PropertyObject *prop0 = ((PropertyObject*)PropLink->e);
       if (0 == strcasecmp ("Resolved",
                   SymbolTable[prop0->GetName()->GetIndex()])) {
            returnval = 1;
            break;
       }
       PropLink = PropLink->succ;
    } while (PropLink != PropHead);
  }
  return (returnval);
}
```

DUObject* SearchDU_generic(NameObject* namedu)

```
{
  if (NULL != namedu) {
  SolarObject *sn = (SolarObject*) AnotherTopNode;
  Link *DULink, *DUHead;
  int ind;
  DUHead = DULink = sn->designunit->head;
  if (DULink != 0){
    DUObject* d= (DUObject*)DULink->e;
    NameObject* n= d->GetName();
    ind = n->GetIndex();
    if (ind != namedu->GetIndex()){
      do{
    DULink = DULink->succ;
    ind = ((DUObject*) DULink->e)->GetName()->GetIndex();
      } while((ind != namedu->GetIndex()) && (DULink != DUHead));
    }
  }
  int ind1 = namedu->GetIndex();
  if (ind == ind1 )     return (DUObject*)DULink->e;
  else return ((DUObject *)GetDUFile_generic(namedu));
  }
}
```

8.2 EDIF Translation methods

extern "C" void PrintEDIFDataStruct()

```
{
  if (TopNode->IsA() == SolarNode) {
    SolarObject *SN = (SolarObject*) TopNode;
    if (SN != 0) SN->EDIFprint();
  }
  else printf("error %d\n", TopNode->IsA());
}
```

void NameObject::EDIFprint()

```
/*
                    +======================+
                    |   Key words Mapping  |
                    +----------------------+
                    |  Solar   |     EDIF  |
                    +----------------------+
                    |    IN    |    INPUT  |
                    |    OUT   |    OUTPUT |
                    +======================+
*/
{
  if (strcmp(SymbolTable[index],"IN") == 0) fprintf(ttout," INPUT ");
  else if (strcmp(SymbolTable[index],"OUT") == 0)
       fprintf(ttout," OUTPUT ");
  else
    {
     if (1 == manipulate_net_name)     // JPS' mail of 13/5/94
       {
         char * net_name;
                net_name = new char  [LEN_NET_NAME];
         static char net_prefix[] = "INIT";     // To check if prefix = "Net_"
         strcpy (net_name, SymbolTable[index]);
         if (strlen (net_name) >= strlen(net_prefix)) {
             for (int i = 0; i < strlen(net_prefix); i++)
                net_prefix[i] = net_name[i];
             net_prefix[strlen(net_prefix)] = '\0';
             if (strcmp("Net_", net_prefix) == 0) {
                for (i = 0; i <= strlen(net_name); i++)
                    net_name[i] = net_name[i+strlen(net_prefix)];
                }
           }
         fprintf(ttout," %s ", net_name);
         manipulate_net_name = 0;
         delete net_name;
       }
```

```
    else fprintf(ttout," %s ", SymbolTable[index]);
   }
}
```

void SolarObject::EDIFprint()

```
{
  edifHead();           // specific edif Header
  if (designunit != 0) {
     designunit->EDIFprintExternal();
     designunit->EDIFprint();
  }
  else printf ("\t:-< WARNING: designunit Missing ***\n");
  fprintf(ttout,")\n");
  DUObject *du = ((DUObject *) this->designunit->head->e);
  char *cell_ref_name;
  cell_ref_name = new char [LEN_CELL_NAME];
  strcpy(cell_ref_name, SymbolTable[du->GetName()->GetIndex()]);
  fprintf(ttout,"(design %s (cellRef %s (libraryRef DESIGNS)))\n",
                      whichEDIF, cell_ref_name);
  fprintf(ttout,")"); // to close edifHead
  fprintf(ttout,"\n");
}
```

void DUObject::EDIFprintExternal()

```
{
  fprintf(ttout,"(external %s (edifLevel %d)\n", libraryRef, edifLevel);
  fprintf(ttout,"(technology (numberDefinition))\n");
  ViewObject *v = ((ViewObject *) this->view->head->e);
  ContentsObject *c = v->GetContents();
  List *inst = c-> instance;
  Link *inst_head, *inst_next;
  inst_head = inst_next = inst->head;
  char cellHistory[LEN_CELL_NAME*MAX_CELLS];
  if (inst_head != 0) {
     do {
     InstanceObject* i = ((InstanceObject *) inst_next->e);
     ViewrefObject *vr = i->GetViewref();
     if (strstr(cellHistory, SymbolTable[vr->GetUnit()->GetIndex()])) {
     inst_next = inst_next->succ;
     continue;
     }
     else {
     FILE *cell;
     char cellFileName[LEN_cellFileName];
     strcpy (cellFileName, libraryRef);
     strcat (cellFileName, "/");
     strcat (cellFileName, SymbolTable[vr->GetUnit()->GetIndex()]);
     strcat (cellHistory,  SymbolTable[vr->GetUnit()->GetIndex()]);
```

```
      strcat (cellHistory, " ");
      strcat (cellFileName, ".solar");
      cell = fopen(cellFileName, "r");
         if (cell == NULL) {
            cout << "\n    :-< File " << cellFileName
              << " is MISSING\t(required by DUObject::EDIFprintExternal)\n";
         }
         else {
            cout << "\n    :-) Parsing " << cellFileName << "\n";
         SolarObject *T = (SolarObject *) TopNode;
         yyparse();
         SolarObject *sn = (SolarObject *) TopNode;
         DUObject *du = ((DUObject *)sn->designunit->head->e);
            fprintf(ttout,"(cell ");
            fprintf(ttout,SymbolTable[du->GetName()->GetIndex()]);
            fprintf(ttout," (cellType GENERIC)\n");
            fprintf(ttout," (view Netlist_representation (viewType NETLIST)\n ");
         ViewObject *vi = ((ViewObject *) du->view->head->e);
            IfaceObject *iface = vi->GetInterface();
            fprintf(ttout,"(interface \n");
         List *port = iface->port;
         Link *port_head, *port_next;
         port_head = port_next = port->head;
         if (port_head != 0) {
         do{
            PortObject *p = ((PortObject *) port_next->e);
            DirObject  *d = ((DirObject *) p->GetDir());
            fprintf(ttout,"(port ");
            fprintf(ttout, SymbolTable[p->GetName()->GetIndex()]);
            fprintf(ttout," (direction");
            if (strcmp(SymbolTable[d->GetName()->GetIndex()],"IN") == 0)
               fprintf(ttout," INPUT");
            else if (strcmp(SymbolTable[d->GetName()->GetIndex()],"OUT") == 0)
               fprintf(ttout," OUTPUT");
            else
               fprintf(ttout," %s ", SymbolTable[d->GetName()->GetIndex()]);
            fprintf(ttout,"))\n");
            port_next = port_next -> succ;
         } while (port_head != port_next);
         }
      fclose(cell);
         fprintf(ttout,")))\n");
         }
      inst_next = inst_next->succ;
      }
      } while (inst_head != inst_next);
   }
   fprintf(ttout,")\n");
}
```

void DUObject::EDIFprint()

```
{
  fprintf(ttout,"(library DESIGNS (edifLevel 0)
    (technology (numberDefinition))\n");
  fprintf(ttout,"(cell ");
  if (name != 0) name->EDIFprint(); fprintf(ttout," ");
  fprintf(ttout,"(cellType GENERIC)\n");
  if (view != 0) view->EDIFprint();
  fprintf(ttout,")\n");
}
```

void List::EDIFprintExternal()

```
{
  Link *r, *f;
  r = f = head;
  if (r != 0){
    do{
      r->e->EDIFprintExternal();
      r = r->succ;
    } while (r != f);
  }
}
```

void List::EDIFprint()

```
{
  Link *r, *f;
  r = f = head;
  if (r != 0){
    do{
      r->e->EDIFprint();
      r = r->succ;
    } while (r != f);
  }
}
```

void ViewObject::EDIFprint()

```
{
  fprintf(ttout,"(view Netlist_representation");
  if (strcmp(viewtype,"") != 0) fprintf(ttout,"(viewType NETLIST)\n");
  if (interface != 0) interface->EDIFprint();
  if (contents != 0) contents->EDIFprint();
  fprintf(ttout,")\n");
}
```

void ContentsObject::EDIFprint()

```
{
  fprintf(ttout,"(contents \n");
  instance->EDIFprint();
  net->EDIFprint();
  fprintf(ttout,")\n");
}
```

void DirObject::EDIFprint()

```
{
  fprintf(ttout,"(direction ");
  name->EDIFprint(); fprintf(ttout,")");
  fprintf(ttout,")");
}
```

void PortObject::EDIFprint()

```
{
  fprintf(ttout,"(port ");
  name->EDIFprint();
  direction->EDIFprint();
  if (1 == comment)     // see JPS's mail 20/8/93
  { fprintf(ttout, "\n");
  fprintf(ttout, "(comment \"Port %s, Array Size is : %d\")",
             comment_port_name, comment_array_index);
  fprintf(ttout, "\n");
  comment = 0;
  }
  fprintf(ttout,"\n");
}
```

void ViewrefObject::EDIFprint()

```
{
  fprintf(ttout,"(viewRef Netlist_representation (cellRef ");
  if (unitname != 0) unitname->EDIFprint();
  fprintf(ttout,") (libraryRef ");
  if (libname != 0) libname->EDIFprint();
  else fprintf(ttout, libraryRef);
  fprintf(ttout," )))");
  fprintf(ttout,"\n");
}
```

void IfaceObject::EDIFprint()

```
{
  fprintf(ttout,"(interface \n");
  port->EDIFprint();
  access->EDIFprint();
  fprintf(ttout,")\n");
```

```
}
```

void NetObject::EDIFprint()

```
{
  strcpy (printed_portRef_instRef, "$ ");     // Initialisation
  is_net_unified = 0;
  int starting_net = 0;
  int stored_Ref_str_portRef_instRef = 0;     // # of ...
  int stored_unified_net_components  = 0;     // # of ...
     char *this_net_name;
     this_net_name = new char [LEN_NET_NAME];
     char *local_net_name;
     local_net_name = new char [LEN_NET_NAME];
     char *unified_net_name;
     unified_net_name = new char [MAX_NETS*LEN_NET_NAME];
     char *Ref_str_portRef_instRef[MAX_PORTREFS];
     for (int k=0; k < MAX_PORTREFS; k++) {
         Ref_str_portRef_instRef[k] = new char [LEN_PORTREF_NAME+LEN_INSTREF_NAME];
     }     // for (int k=0;
     char *unified_net_components[MAX_NETS];
     for (k=0; k < MAX_NETS; k++) {
         unified_net_components[k] = new char [LEN_NET_NAME];
     }     // for (int k=0;
     char *list_portRef_instRef;
     list_portRef_instRef = new char [MAX_NETS*MAX_PORTREFS*
       (LEN_PORTREF_NAME+LEN_INSTREF_NAME)];
     char *local_list_portRef_instRef;
     local_list_portRef_instRef = new char [MAX_PORTREFS*
           (LEN_PORTREF_NAME+LEN_INSTREF_NAME)];
     strcpy (list_portRef_instRef, "$ ");
     strcpy (this_net_name, SymbolTable[this->GetName()->GetIndex()]);
     strcpy (unified_net_name,this_net_name);
     strcpy (unified_net_components[stored_unified_net_components++],
             this_net_name);
  ContentsObject *cO = (ContentsObject *) this->parent;
  List *net = cO->net;
  Link *net_head, *net_next;
  net_head = net_next = net->head;
  if (net_head != 0) {
     do {
     NetObject *nO     = ((NetObject *) net_next->e);
     if (strcmp (this_net_name,
           SymbolTable[nO->GetName()->GetIndex()]) == 0) {
             int i = 0;
             if (nO -> joined != 0) {
                 JoinedObject *jO = nO->GetJoined();
                 List *pref = jO->portref;
                 Link *pref_head, *pref_next;
```

```
              pref_head = pref_next = pref->head;
              if (pref_head != 0) {
               do {
                 PortrefObject *pr0 = ((PortrefObject *) pref_next->e);
                 InstrefObject *ir0 = pr0->GetInstref();
                 if (ir0 != 0) {
                   strcpy(Ref_str_portRef_instRef[i], "#");
                   strcat(Ref_str_portRef_instRef[i],
SymbolTable[pr0->GetValue()->GetIndex()]);
                   strcat(Ref_str_portRef_instRef[i], "#");
                   strcat(Ref_str_portRef_instRef[i],
SymbolTable[ir0->GetName()->GetIndex()]);
                   strcat(Ref_str_portRef_instRef[i], "$ ");
                     stored_Ref_str_portRef_instRef = i;
                     if (strstr(printed_portRef_instRef,
                         Ref_str_portRef_instRef[i])){
                         is_pR_iR_found = 1;
                     }
                   i++;
                 }     // if (ir0 != 0)
                 pref_next = pref_next -> succ;
               } while (pref_head != pref_next);
               }     // if (pref_head != 0)
             }          // if (n0 -> joined != 0)
       starting_net = 1;
  }     // if (strcmp (this_net_name
  else {
       if ( 1 == starting_net)  {
       strcpy (local_list_portRef_instRef, "$ ");
       strcpy (local_net_name,
                     SymbolTable[n0->GetName()->GetIndex()]);
  if (n0 -> joined != 0) {
      JoinedObject *j0 = n0->GetJoined();
      List *pref = j0->portref;
      Link *pref_head, *pref_next;
      pref_head = pref_next = pref->head;
      if (pref_head != 0) {
       do {
         PortrefObject *pr0 = ((PortrefObject *) pref_next->e);
         InstrefObject *ir0 = pr0->GetInstref();
         if (ir0 != 0) {
           strcpy(str_portRef_instRef, "#");
           strcat(str_portRef_instRef, SymbolTable[pr0->GetValue()->GetIndex()]);
           strcat(str_portRef_instRef, "#");
           strcat(str_portRef_instRef, SymbolTable[ir0->GetName()->GetIndex()]);
           strcat(str_portRef_instRef, "$ ");
           strcat(list_portRef_instRef,str_portRef_instRef);
           strcat(local_list_portRef_instRef,str_portRef_instRef);
         }     // if (ir0 != 0)
```

```
          pref_next = pref_next -> succ;
        } while (pref_head != pref_next);
        }     // if (pref_head != 0)
      }     // if (n0 -> joined != 0)
      for (int i = 0; i <= stored_Ref_str_portRef_instRef; i++) {
         if (strstr(local_list_portRef_instRef, Ref_str_portRef_instRef[i])){
      strcat(unified_net_name, "_");
      strcat(unified_net_name, local_net_name);
      strcat(unified_net_components[stored_unified_net_components++],
             local_net_name);
        }     // if (strstr(local_list_portRef_instRef
       }     // for (int i = 0;
    }     // if ( 1 == starting_net)
   }     // else
      net_next = net_next->succ;
      } while (net_head != net_next);
   if ((strcmp(this_net_name,unified_net_name) != 0) &&
      (! strstr(marked_nets,unified_net_name))) {
   strcat(marked_nets, unified_net_name);
   strcat(marked_nets, "_");
      } // if (strcmp(this_net_name,unified_net_name)
 }     // if (net_head != 0)
      for (int i = 0; i <= stored_Ref_str_portRef_instRef; i++) {
        if (strstr(list_portRef_instRef, Ref_str_portRef_instRef[i])){
           is_net_unified = 1;
           break;
        }     // if (strstr(list_portRef_instRef
      }     //for (int i = 0;
      if (0 == is_net_unified) {/* action /* } // if (0 == is_net_unified)
 delete local_net_name;
 delete list_portRef_instRef;
 for (int m=0; m < MAX_PORTREFS; m++) {
     delete Ref_str_portRef_instRef[m];
 }     // for (int m=0;
manipulate_net_name = 1;
char * this_net_string;
       this_net_string = new char  [LEN_NET_NAME];
strcpy(this_net_string, "#");
strcat(this_net_string, this_net_name);
strcat(this_net_string, "$ ");
if (! strstr(printed_nets, this_net_string)){
fprintf(ttout,"(net ");
if (0 == is_net_unified) {     // Simple Net
     name->EDIFprint();
     fprintf(ttout,"\n");
     if (joined != 0) joined->EDIFprint();
   strcat(printed_nets, "#");
   strcat(printed_nets, this_net_name);
   strcat(printed_nets, "$ ");
```

```
  }
  else {                          // Unified Net
       printName(unified_net_name);
       ContentsObject *cO = (ContentsObject *) this->parent;
       List *net = cO->net;
       Link *net_head, *net_next;
       net_head = net_next = net->head;
       if (net_head != 0) {
       do {
          NetObject *nO     = ((NetObject *) net_next->e);
     strcpy (this_net_name, SymbolTable[nO->GetName()->GetIndex()]);
          for (int j = 0; j < stored_unified_net_components; j++) {
          if (strcmp (this_net_name, unified_net_components[j]) == 0) {
             if (nO -> joined != 0) {
                if (1 == is_net_unified) {
                    fprintf(ttout,"\n");
                  fprintf(ttout,"(joined\n");
                  is_net_unified = 2;
                }
                  nO -> joined -> EDIFprint();
                  strcat(printed_nets, "#");
                  strcat(printed_nets, SymbolTable[nO->GetName()->GetIndex()]);
                  strcat(printed_nets, "$ ");
                }          // if (nO -> joined != 0)
          }
          }
          net_next = net_next->succ;
       } while (net_head != net_next);
       fprintf(ttout,")\n");
          }
  }
  fprintf(ttout,")\n");
  } // (strstr(printed_nets, this_net_string))
  delete this_net_string;
}
```

void JoinedObject::EDIFprint()

```
{
  if (0 == is_net_unified) fprintf(ttout,"(joined\n");
  if (portref != 0) portref->EDIFprint();
  if (accessref != 0) accessref->EDIFprint();
  if (0 == is_net_unified) fprintf(ttout,")\n");
}
```

void PortrefObject::EDIFprint()

```
{
  if (0 == is_pR_iR_found) {
  fprintf(ttout,"(portRef ");
```

```
  if (value->IsA() == NameNode) value->EDIFprint();
  if (instref != 0) instref->EDIFprint();
  fprintf(ttout,")");
  fprintf(ttout,"\n");
  strcpy(str_portRef_instRef, "#");
  strcat(str_portRef_instRef, SymbolTable[this->GetValue()->GetIndex()]);
  }
}
```

void InstrefObject::EDIFprint()

```
{
  if (0 == is_pR_iR_found) {
  fprintf(ttout,"(instanceRef ");
  name->EDIFprint();
  if (instref != 0) instref->EDIFprint();
  fprintf(ttout,")");
  strcat(str_portRef_instRef, "#");
  strcat(str_portRef_instRef, SymbolTable[name->GetIndex()]);
  strcat(str_portRef_instRef, "$ ");
  strcat(printed_portRef_instRef, str_portRef_instRef);
  }
}
```

void InstanceObject::EDIFprint()

```
{
  fprintf(ttout,"(instance ");
  name->EDIFprint(); fprintf(ttout,"\n");
  if (viewref != 0) viewref->EDIFprint();
}
```

void ArrayObject::EDIFprint()

```
{
  comment = 1;
  name->EDIFprint();
  strcpy(comment_port_name, SymbolTable[name->GetIndex()]);
  value->EDIFprint();
}
```

void IntvalObject::EDIFprint()

```
{
  if (1 == comment) comment_array_index = value;
  else fprintf(ttout," %d ", value);
}
```

RESUME:

Le sujet traité dans cette thèse, concerne les liens entre la synthèse de haut niveau (HLS: High Level Synthesis) et la synthèse au niveau transfert de registres (RTL: Register Transfer Level). Il s'agit d'une adaptation de l'architecture résultat de la synthèse de haut niveau par transformation en une description (au niveau) RTL acceptée par les outils industriels actuels. Les objectifs visés par cette transformation, sont: accroître la flexibilité et l'efficacité, permettre la paramétrisation de l'architecture finale. A partir d'une description comportamentale décrite dans un language de description de materiel (la synthèse de haut niveau) génère une architecture au niveau transfert de registres, comprenant un contrôleur et un chemin de données. Le contrôleur et le chemin de données peuvent être synthétisés par des outils de synthèse RTL et logique existant pour réaliser un ASIC ou un FPGA. Cependant, pour des raisons d'efficacité, il est préférable de synthétiser le chemin de données par un compilateur de chemin de données. Nous allons dans un premier temps concevoir une méthode que nous appelerons personnalisation. Elle permet aux concepteurs d'adapter l'architecture générée aux outils de synthèse RTL et à toute structure particulière requise. Ensuite, nous définirons une méthode appelée Décomposition. Cette dernière fournira un moyen de décomposer un chemin de données en plusieurs sous chemins de données réguliers, pouvant être synthetisés de manière efficace par un compilateur de chemin de données. Enfin, nous présenterons la génération de chemins de données génériques, destinés à la réalisation d'architectures paramétrables au niveau RTL. Cet algorithme a été implanté dans le generateur de code VHDL à partir de la structure de données intermédiaire utilisée par AMICAL, un outil de synthèse de haut niveau.

Mots Clés: Synthèse, Personalisation, Decomposition, Tranductions.

ABSTRACT:

This thesis work addresses the subject of linking High Level Synthesis (HLS) of VLSI with the Register Transfer Level (RTL) synthesis prevailing to be the current Industrial practice. Starting from the HLS results, we generate quality RTL specifications equivalent of the initial behavioral specification. The generated results aim at an improved flexiblity, efficiency and parameterization from a designer's view point in terms of the final architecture. A HLS starts from a behavioral description in a Hardware Description Language (HDL) such as the VHDL, performs scheduling, allocation to generate a RTL architecture based on a controller and a datapath. Both controller and datapath can be synthesized by the logic synthesis tools to realize an Application Specific Integrated Circuit (ASIC) or a Field Programmable Gate Array (FPGA). However, for efficieny reasons, it is preferable to synthesize the datapath using a datapath compiler. Further, the architecture obtained as a result of HLS can be parameterized. We develop a flexible Personalization method to allow the designers not only to add the information related to the synchronization at the RT level, but also to mix in a high level description, both synthesizable and un synthesizable parts. Next we define a method known as a Decomposition to transform an available HLS datapath into an interconnection of several regular datapaths and a glue logic. All the extracted regular datapaths can undergo efficient synthesis by a datapath compiler. Finally, we present the delivery of generic datapaths supporting parameterizable architectures at the RT level. The idea is incorporated into a VHDL translator from the intermediate data structure used by AMICAL, a HLS tool.

Key Words: Synthesis, Personalization, Decomposition, Translations.

type, or another derived class of the object declared in the database or a list.

The four attributes public to a generic object are the following:

Type	Attribute
Object	parent
List	property
List	toolinfo
Integer	nodenum

Parent is a pointer to a precedent node in the circuit representation (hierarchy). *Property* and *Toolinfo* are the lists that may contain the information added by the user. *Nodenum* is an integer that will be affected by a value corresponding to its position in the actual representation.

A list is a structure that contains the *links* in a doubly linked circular manner. A *link* is a structure of type class containing the following public attributes:

Type	Attribute
Link	succ
Link	pred
Object	e

A (circular, doubly linked) link contains pointers to links for the predecessor and successor within a given list. In addition, it contains a pointer to an element of type *generic*, namely, the *Object*. Because, all the possible nodes are derived from an *Object* type, links are equally generic. Any *Object* present within the database can be accessed using the link pointers. A list contains two attributes:

Type	Attribute
ObjectType	ListType (Private)
Link *	head

The first attribute is private and is initialized while the list is created. Other links to the list can contain only the objects of the same type which is given by a *ListType*. The second attribute is a pointer to the first link in the list.

When a Solar file is compiled, the objects, links and lists are interconnected in order to generate a directed graph representing the circuit as shown in *Figure* 6.2.

The root node is always an instantiation of the type *SolarObject*. Starting from this root node, a path towards any other node within the graph may be created. Similarly, a path exists between all the nodes of the graph and the root node. The connections are established using the attributes, links and parents. An Object may have several successors, however, only one parent.

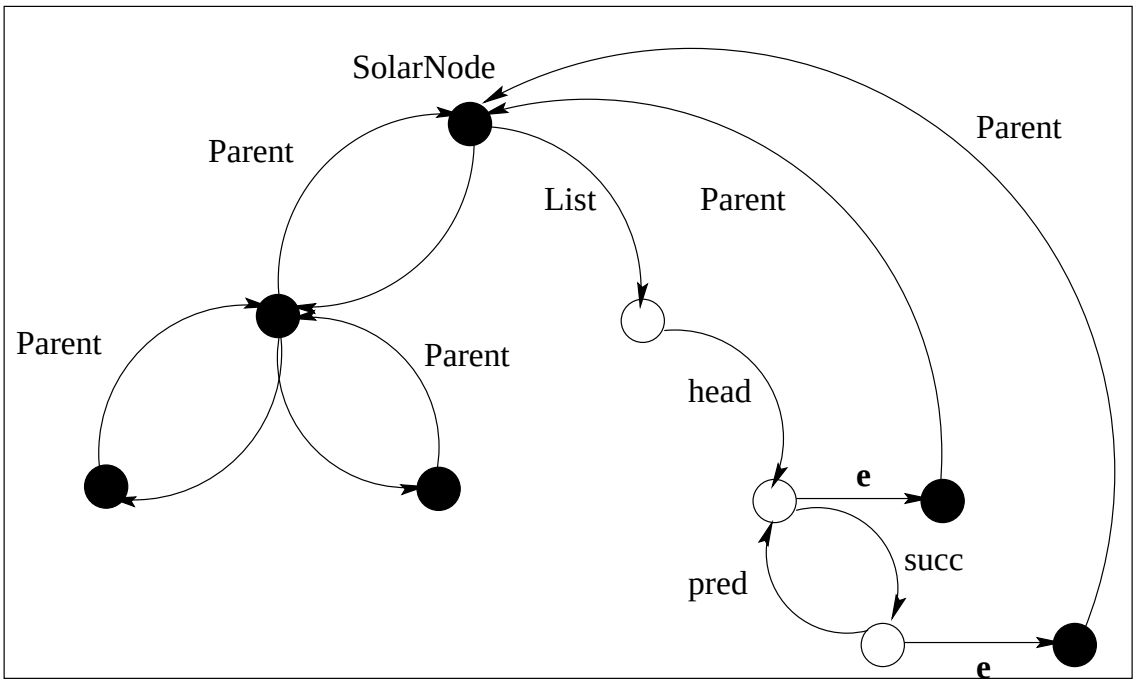


Figure 6.2: Internal structure of Solar

Object families

The rest of the Solar environment is composed of objects representing the different derivations, either direct or indirect, of the generic type of *Object*. Each object contains a list of attributes and a set of operations that manipulate the attributes. There are actually, more than about seventy different objects present in the Solar environment. These objects may be found in the annex-B of [O'B93].

The principal classes of objects, together with the hierarchy of their derivations, are shown in *Figure* 6.3.

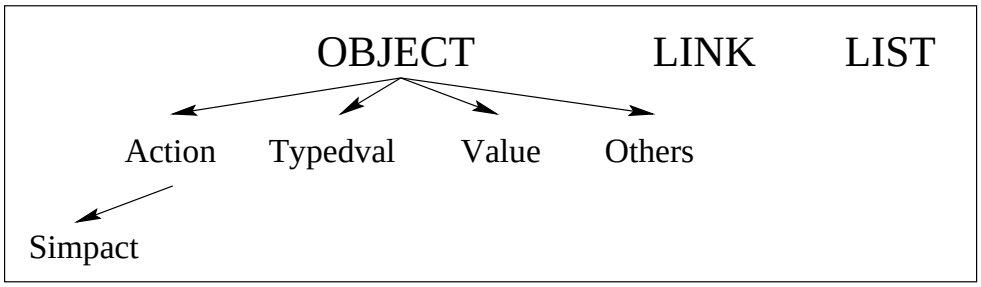


Figure 6.3: Solar derived classes

The three fundamental objects, already described, are *Object, link* and *list*. Neither the class *link*, nor the class *list* contains their derived classes. The *Object* class contains several derived classes, three of which forming their object families. The class *Action* contains the nodes represnting the Solar concepts of *ParAction* (the parellel actions),

SeqAction (the sequential actions), etc. It also contains a derived class, *Simpact* (the simple actions), containing the objects for representing the operations such as *if, while, wait,* etc. The family *Typedval* contains the classes for types. The family *Value* contains the classes of *operands* such as the tables and identifiers. The other classes, directly derived from the *Object* class, includes the units, namely, CU (the communication unit), FU (the functional unit) and ChU (the channel unit), as well as certain objects necessairy to complete the environment.

Compilation of Solar

The compiler or analyzer of Solar utilizes the tools lex [1] and yacc [2] in order to read and analyze the input files. During the compilation of a Solar file, the nodes corresponding to the actual instructions are created and added to the object oriented data structure. Nearly all the Solar key words are treated during the creation of a new node. The information are added to this node until the end of the instruction (the closing bracket ')' is encountered to be the next token). Then, the node is attached to an appropriate attribute of its parent. If this attribute is a list, then the node is inserted to the head of list (The lists are doubly linked and circular in order to facilitate further searches in the data structure). The compilation is briefly described for a segment of Solar code as follows.

```
(DESIGNUNITserver              ;Create DU, Name nodes;
                                 DU− >SetName(Name)
    (VIEW behavior             ; Create View, Name nodes;
                                 View− >SetName(Name)
      (CONTENTS                ; Create ContentsNode
        (STATETABLE server     ; Create ST, Name nodes;
                                 ST− >SetName(Name)
          (STATE idle          ; Create NxtState, Name nodes;
                                 NxtSt− >SetName(Name),
                                 St− >SetNxtSt(NxtSt)
          )                    ; ST− >state− >Append(ST)
        )                      ; Contents− >ST− >Append(ST)
      )                        ; View− >SetContents(Contents)
    )                          ; DU− >View− >Append(View)
)                              ; Solar− >DU− >AppendDU
```

More than ninety tokens may be identified by the lexical analyzer utilized by Solar. In the above description of Solar, it is assumed that the root node, namely, *SolarNode*, already exists. The parent of the root node is a null pointer.

[1] lexical analysis program generator of Unix.

[2] yet another compiler-compiler: a parsing program generator of Unix.

When the token corresponding to the key word *DesignUnit* is encountered, a design unit node (*DUNode*) is created. A node of this type contains an element of the type *DUObject*. This element is a directly derived class of *Object*. An object of type *DUObject*, as well as all the attributes (either derived or underived) and their initial values, are presented below:

```
DUObject
    Object      *parent;   //Derived attribute, initial value = TopNode *
    List        *property; //Derived attribute, initial value = new List(property)
    List        *toolinfo; //Derived attribute, initial value = new List(toolinfo)
    int         nodenum;   //Derived attribute, initial value = 0
    NameObject  *name;     //Derived attribute, initial value = null
    List        *view;     //Derived attribute, initial value = new List(ViewNode)
```

The grammar demands that four out of the six fields for a design unit be furnished their initial values. They are, namely, *parent, nodenum, view* and *name*. *parent* is a pointer to the predecessor of the node. Whenever a node is created, it is placed on a stack of nodes created but not yet filled. The global pointer, *TopNode*, has the address of the last node. As well, the field *parent* takes the value of *TopNode*, and *TopNode* takes the address of new node (in this case, the *DUNode*). The field *nodenum* has a unique value for this node in the representation.

All the lists are created during the compilation (including the *property* and the *toolinfo*). By the way, none of the lists contain their links yet, at the time of creating the the lists of nodes. The attribute *head* is a null pointer (there are not yet the links). However, the attribute *ListType* is initialized to an appropriate type of the node. For example, the creation of a new list for the attribute *view* of node *DUNode*, would be of the form:

view = new List(ViewNode);

The node created next to *DUNode* is a node of the identifier *NameNode*. It corresponds to the name of the designunit. When this node is created, it becomes the new *TopNode*. Instead of storing a string of characters for each node of identifier created, one uses a data structure called a symbol table (*SymbolTable*). The symbol table keeps only a single copy of each identifier, sometimes its type. An identifier node contains two attributes of type integer; one has the address of the character string in the symbol table (*index*) and the other corresponds to the object type of the parent (*idtype*). The advantage of storing the character string in this fashion is indicated in *Figure* 6.4.

Within the code segment, there are three entities having the same name. Instead of storing three character strings, only a single copy of the identifier is stored within the symbol table. For each identifier, recognized during an analysis, the symbol table is searched in order to verify if the identifier already exists. If it is not the case, it is

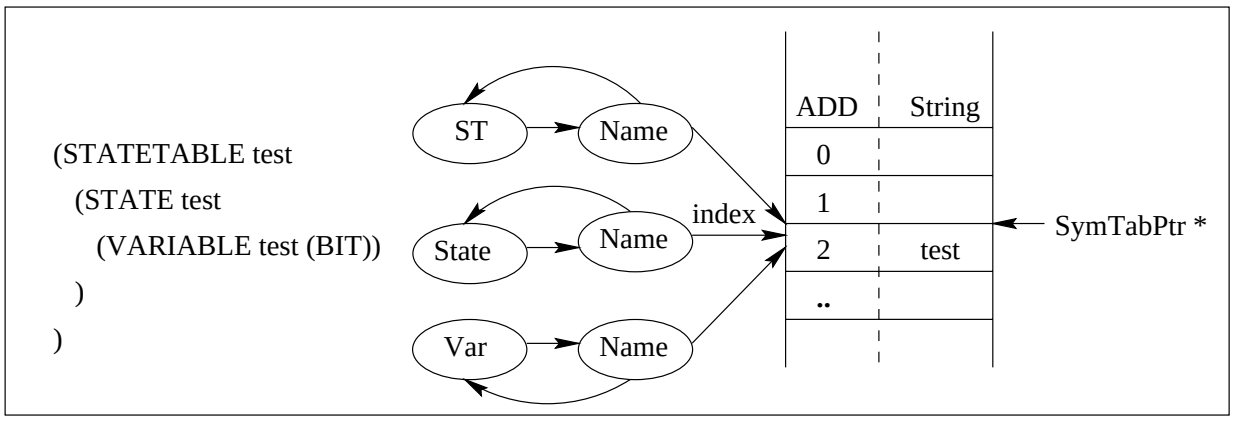


Figure 6.4: Solar symbol table

inserted into the table and an integer value corresponding to its address is stored to the attribute *index* of the identifier node (*TopNode− >index*). If one has to know this identifier type, the data structure is traversed to search the type of *parent* of the identifier node:

$$TopNode- > SetID(TopNode- > parent- > IsA());$$

SetID() and *IsA*() are the functions from the procedural interface (the next section).

Once this identifier node is filled, it is removed out of the stack. *TopNode* becomes the parent of this node (*TopNode = TopNode− >parent*). In this example, *TopNode* is newly the design unit node. The compilation is terminated once there are no more nodes left out on the stack.

Procedural Interface

To access and manipulate the attributes, Solar provides a set of functions, which is named as the procedural interface to the data base. These functions are declared within the classes themselves. These are known as the member functions. The member functions declared as virtuel are automatically inherited by the derived classes. The member functions are provided in [O'B93] annex-B.

The functions that are common to all the objects are declared as the member functions of the class *Object*. They are the following:

virtual	ObjectType	**IsA()**
virtual	ObjectType	**IsBase()**
virtual	void	**print**()
virtual	int	**MyNum()**

The function *IsA()* is utilized for identifier of the node type actually assigned. Similarly, the function *IsBase()* gives the base type of which the actual node has been

derived if it is not directly derived of *Object*; Else, it gives the same result as of *IsA()*. The function *MyNum()* gives the node identity number within the representation of a circuit. The function *print()* displays, on the standard terminal, the values of each attriute of the actual node.

The attributes that are declared as the lists contain automatically the following functions:

```
void      append(Object *a)     // Add a link
int       length()              // Number of links
Object *  RemoveHead()          // Remove the first element
Object *  RemoveTail()          // Remove the last element
void      print()               // display the contents of the list
```

These functions explain how to traverse a Solar data structure. A description of the function *RemoveTail()*, used for removing the last element of a list is given below:

```
Object *List::RemoveTail()
/* This function removes the last element of a list */
{
   if (head == 0)    // No traversal, if list is found empty
   {
          print ("Error:  List empty
n");
          return 0;
   }
   else
   {
   {
          while (!t->succ == head) t = t->succ; // Traverse list until end
          res = t->e;                           // Get last element
          t->succ->pred = t->pred;              // Modify pointers
          t->pred->succ = t->succ;
   }
          delete t;
          return res;
   }
}
```

Other functions calling functions such as the *RemoveTail*, are described at a higher level. Each object contains other member functions for reading and affecting the attributes. For example, there exists a function associated to each object, which is called *vhdl()*. This function translates the contents of nodes to instructions equivalent

in the VHDL language. Similarly, another high level function called *edif()* does the translation from Solar to EDIF. A part of the declaration of a design unit object is given below:

```
class DUObject : public Object
{
      NameObject        *name;
      List              *view;        //All the member functions of list applicable.
public:
      DUObject();                     //Constructor functon; Initialize Attributes.
       DUObject();                    //Destructor functon.
      void              print();
      void              vhdl();
      void              edif();
      void              Pre_vhdl();
      void              check();
      void              partition();
      ObjectType        IsA()         {return DUNode;};
      ObjectType        IsBase()      {return DUNode;};
      void              SetName(NameObject *n)        {name = n;};
      NameObject        *GetName                      {return name;};
}
```

Being a structured representation, it is quite simple to add new functions to the procedural interface which in turn facilitates the extension of Solar.

6.2 Model

The architecture generated by AMICAL is composed of a *controller*, a *datapath*, an *interface* from the controller to the datapath, an *interface* from the datapath to the controller and a *circuit* that interconnects the above 4 blocks. Each model can be tailored by the user for a target architecture using the personalization step.

6.2.1 Controller

Controller is a completely specified [3] sequential *Finite State Machine* (FSM) of type Mealy.

Table 6.1 represents how the FSM variables are represented in the Solar language for an example containing two states:

[3] A macine whose next state function and/or the output function are not completely specified is called an *incompletely specified machine*.

FSM representation	Significance	Solar example
I	Input Ports	(direction IN)
S	States	(StateList S1 S2)
O	Output ports	(direction OUT)
F_{nxt}	State Assignment	(NextState S2)
F_{out}	Output Assignment	(Assign Pout '1')

Table 6.1: FSM representation in Solar

I is a finite non empty set of input symbols $i_1, i_2, ..i_m$.
S is a finite non empty set of states $s_1, s_2, ..s_n$.
O is a finite non empty set of output symbols $o_1, o_2, ..o_k$.
F_{nxt} is the next state function, which maps S x I -> S.
F_{out} is the output function, which maps S x I -> O.

The controller in Solar is represented using: an interface, I_{face}, such that I_{face} = I $\bigcup$ O and a statetable (ST) containing a finite non empty set of states (S). The controller in VHDL may be represented using: an entity, E, such that E = I $\bigcup$ O, an architecture, composed of 2 processes, P_1 and P_2. The process (P_1) contains a finite non empty number of mutually exclusive case statements, each of which corresponding to a particular state out of S. The process (P_2) is used to synchronize the movement of the FSM going from one state to another. The synchronization can be customized by the user suiting the target architecture.

The significant parts of the Controller Solar description is given below:

```
(SOLAR AMICAL_for_Controller
    (DesignUnit gcd_control
        (View AbstractArchitecture (ViewType " behavior")
                                   (Property arch_dp MUX)
            (Interface
                (Port start (Direction IN) (Bit) (Property PortType CONTROL))
                ...
                (Port (Array FLAG_x 8) (Direction IN)
                                       (Property PortType CONTROL)
                                       (Property Resolved BusX))
            ) ; Interface
            (Contents
                (StateTable Controller
```

```
                    (State S1
                        (Case
                            (Alt ( & ( = start '1') True )
                                (Assign CTRL_Sel_dout '1')
                                (Assign x_1_dout_Sel '0')
                                (NextState S2)
                    ) ; State S1
                    ...
                    (StateList S1 S2 S5 S3 S4)
                ) ; StateTable
            ) ; Contents
        ) ; View
    ) ; DesignUnit
) ; SOLAR
```

6.2.2 Datapath

Datapath (DP) is characterized by a 6-tuple described as follows:

DP = (FU, SU, CU, EU, IU, Ω)
where,
FU is a set of functional units $fu_1, fu_2, ..fu_n$ such that: n >= 0.
SU is a set of storage units $su_1, su_2, ..fu_m$ such that: m >= 0.
CU is a finite non empty set of (internal) communication units
$cu_1, cu_2, ..cu_k$ such that: k >= 0.
EU is a finite non empty set of external communication units
$eu_1, eu_2, ..eu_j$ such that: j >= 0.
IU is a finite non empty set of interconnection units
$iu_1, iu_2, ..iu_x$ such that: x >= 0.
Ω is a Model of data transfer between the datapath and controller.

The Model of data transfer, Ω, includes a set of instructions to define the dependancies between the sources and the destinations for the transfer operations and a set of generation rules to define the realization of a transfer. More information on the transfer models are available in [Din96].

Each of the 5 units, namely the: FU, SU, CU, EU and IU can be either defined in an abstract manner, i.e, in the form of a block box or containing more information such as for a performance evaluation. The ensemble of the above 5 units shall be henceforth referred as *Components*. The use of a component within a datapath is termed as an *Instance* and an interconnection between 2 or more instances is called a *Net*.

6.2.3 Interface

The definitions of the interfaces between the controller and the datapath, namely, from the controller to the datapath and vice versa reduce to the definition of Datapath. However, all the components are of type SU, the storage unit.

6.2.4 Circuit

Circuit, CKT, interconnects all of the above mentioned blocks, namely: the controller, the datapath and the Interface cells.

6.2.5 Translation Issues

In this subsection, we shall discuss particular issues that concern the linking of the high level synthesis in Solar to VHDL models for the Register Transfer Level synthesis.

Use of Solar Properties

A certain number of Solar properties serve in the translation process. Some of the frequently used properties are listed below:

1 **PortType** is used to classify the ports. It indicates whether a port is of type *Control, Data* etc. Each data port is qualified to be of type either Generic or Non-Generic.

2 **Resolved** indicates if a port needs to use a *resolved* or *unresolved* data type. In VHDL, a port of type resolved may be simultaneously affected by several drivers of same or different values.

3 **Width** is used to indicate the size of a port. It is always a positive integer. A port of width 1 may be either a bit or an array of size 1 (0 to 0). The property width is used in the generation of a VHDL datapath containing components having *generic*s for the port sizes.

4 **ConstantValue** is a property to determine the constant value content of a ConstantRegister and thus used for its parameterization.

Options of Translation

Certain options provided during the translation enables the user to incorporate the information specific to lower level synthesis.

1 **Library** : VHDL is a strongly type oriented language; The standard data types and operations are usually contained in the *package*s of *library*(ies). During a translation, the user has the options to include a library of his (her) choice. We compare using *Figure* 6.5, the use of two libraries used by the design community.

```
library IEEE;
  use IEEE.Std_Logic_1164.all;
  use IEEE.Std_Logic_ARITH.all;
  use IEEE.Std_Logic_UNSIGNED.all;

  -- use IEEE.Std_Logic_SIGNED.all;

  -- synopsys_synthesis_off
  -- use IEEE.Std_Logic_MISC.all;
  -- use IEEE.Std_Logic_TEXTIO.all;
  -- synopsys_synthesis_on
```

a) Library : IEEE

```
library mvl_7; -- Synopsys
  use mvl_7.TYPES.all;
  use mvl_7.ARITHMETIC.all;

  -- use mvl_7.ARIHSUPP.all;
  -- synopsys_synthesis_off

  -- use mvl_7.TEXTIO.all;
  -- synopsys_synthesis_on
```

b) Library : (synopsys) mvl_7)

Figure 6.5: Choice of VHDL library

2 **Index Direction** : In Solar, there is no way to qualify an array data in terms of an increasing or a decreasing index. An increasing index is the one which begins by the LSB (Lowest Significant Bit) and increases until the MSB (Most Significant Bit). A decreasing index is its inverse. VHDL permits this classification using the "to" and "downto" keywords as shown in *Figure* 6.6 (b). We introduced 2 translation options, namely "to" and "downto" to achieve the respective translation of a desired index direction.

```
(port (array P1 16) (direction inout) ... )
```

a) Solar Example

```
port (p1 : inout Std_Logic_Vector (0 to 15));  -- Port of increasing index.
port (p1 : inout Std_Logic_Vector (15 downto 0));  -- Port of decreasing index.
```

b) VHDL Examples

Figure 6.6: Choice of VHDL Index direction

6.3 Architecture Translation in VHDL

In this section, we explain the translation procedures required by different units of the architecture generated by Amical.

6.3.1 Controller

The controller available after a high level synthesis is a *designunit* having a *viewtype* of *behavior*. It contains a solar interface and a contents; The interface contains ports while the contents contain a statetable. The organization of the statetable of the controller within the contents of SOLAR is given below:

- solar_contents_description ::= (**Contents** state_table_description)
- solar_state_table_description
  ::= (**StateTable** identifier states_description statelist_description)
- states_description ::= (**State** identifier case_description)
- case_description ::= (**Case** alternative_description)
- alternative_description
  ::= (**Alt** condition boolean) assign_description nxt_state_description)
- condition ::= (operator identifier value)
- assign_description ::= (**Assign** identifier value)
- nxt_state_description ::= (**NextState** identifier)
- statelist_description ::= (**StateList** identifier_list)
- identifier_list ::= (identifier .. identifier)

After, translation, the elements of the resulting VHDL code are: the declaration and use of user specified library(ies), an entity which is translation of the interface, an architecture containing 2 processes: the process FSM implements the state table, while the second process updates the current_state signal in a synchronized manner. *Figure* 6.7 explains the translation.

6.3.2 Datapath

The *datapath* available after a high level synthesis is a *designunit* having a viewtype of structure. It contains a solar *interface* and a *contents*; The interface contains ports while the contents contain a set of instances and nets. Examples for 2 instances (Mux_2 and ConstReg) and a net interconnecting the above 2 instances are shown below:

```
(Instance x_1 (ViewRef Implementation Mux_2)
    (PortInstance in1 (Property PortType DATA) (Property WIDTH 1))
    (PortInstance in1 (Property PortType DATA) (Property WIDTH 1))
```

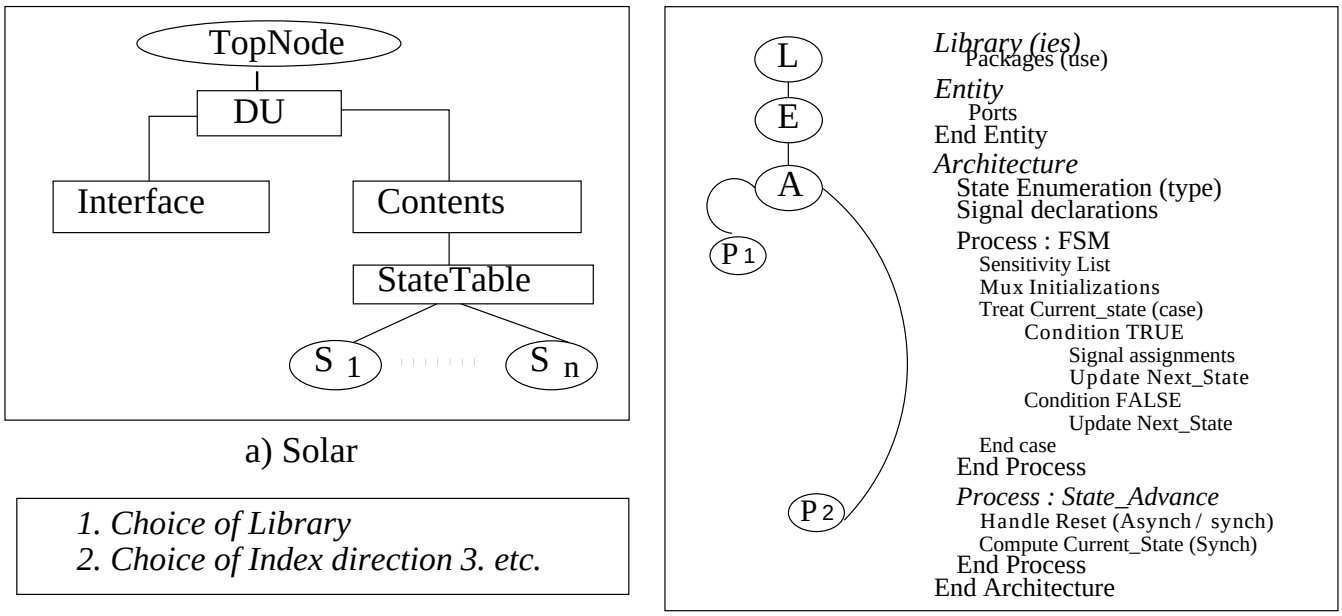


Figure 6.7: Controller translation from Solar to VHDL

```
    (PortInstance out1(Property PortType DATA) (Property WIDTH 1))
    (PortInstance Sel (Property PortType CONTROL))
)
(Instance C0 (ViewRef Implementation ConstReg)
    (PortInstance Data_OUT
                    (Property PortType DATA) (Property WIDTH 8))
)
(Net Net_1          (Joined
                    (PortRef in1            (InstanceRef x_1))
                    (PortRef Data_OUT       (InstanceRef C0))
)
```

After translation, the elements of the resulting VHDL code are: the declaration and use of user specified library(ies), an entity which is a translation of the interface, an architecture containing the component declarations corresponding to the instances present in the initial description and their instantiatiations. The latter is performed by analyzing the nets that are present. *Figure* 6.8 explains the translation.

The instances present in the datapath are either of standard components such as the Functional Units (*FU*), Storage Units, Switching Units, Synchronization Units, External Communication Units, Interface Units etc. (or) of user defined application specific complex components.

Each of the component could be either of type *generic* or *non_generic*. A generic

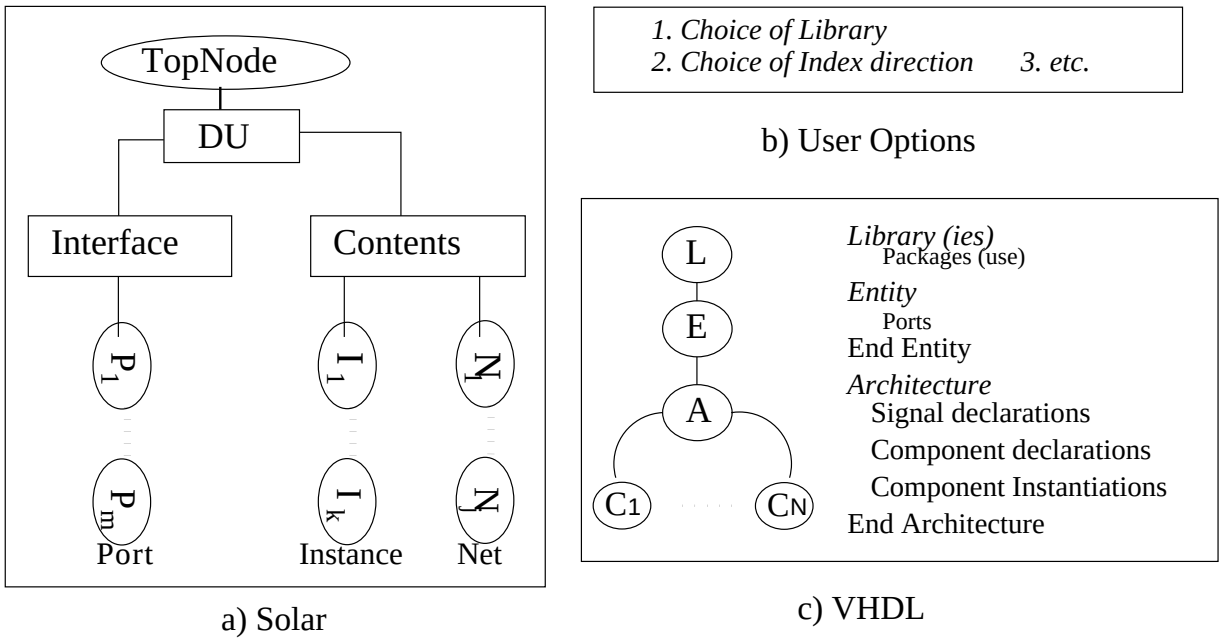


Figure 6.8: Datapath translation from Solar to VHDL

component is one which contains at least one generic value in addition to its ports. The generic values may be used either to parameterize the component (example: ConstantValue of a ConstReg) itself, or its port widths, or both. The datapath can be accordingly translated in 2 modes: generic and non-generic; the choice is left to the user. With the choice of generic mode, it is also possible to use some of the components in a non generic mode, by simply adding a *DATA_NON_Generic* property to all of its ports of portType Data. Such an use is desired in component reuse of already designed and existing macro blocks, whose interface can not be altered by the user. (example: A 32-bit DSP).

6.3.3 Circuit

The Circuit combines the four above mentioned units by interconnection. The interface (ports) of circuit matches the initial high level specification entity and thus permits the use of same test bench for validation by simulation. The contents are the instances of the 4 units and the nets to interconnect them. The organization of the circuit is shown using *Figure* 6.9 in a validation context, wherein, the two simulations are matched for equality.

6.4 Regular Datapaths Translation in EDIF

In this section, we present the translation procedures required by the datapath of the architecture after undergoing a datapath decomposition (Chapter 5).

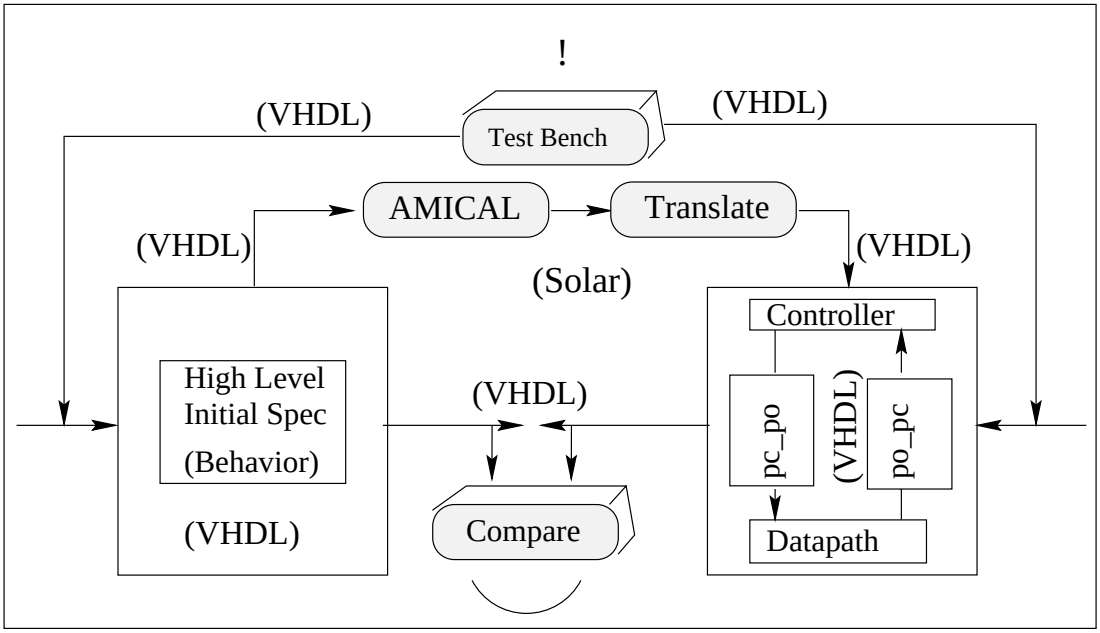


Figure 6.9: Circuit translation from Solar to VHDL and Validation

6.4.1 Datapath

The format of Solar containing the datapath has a lot of similarities with that of EDIF. However, each EDIF description is a parameterizable regular datapath, which means, the ports contained in the EDIF description are all of type *bit* and do not require any 'size' (or) 'width' parameter. Unlike a solar description, which may be abstract, containing generic components, an EDIF description is ready for direct physical implementation in a given technology. The cells used from the technology library are grouped into a list called 'external library'. The datapath design is stored into another list called 'library' containing the Solar Datapath, expressed in terms of 'external' cells.

Genration of an external list

The external list is defined as a list resulting after extracting the unique number of components (viewRef) used by the datapath. Only the interface of each component is needed. The implementation of the component is hidden using a libraryRef name. A part of an external list using a component is shown below:

```
(external DPLIB (edifLevel 0)
(technology (numberDefinition))
          (cell DPINV (cellType GENERIC)
                (view Netlist_representation (viewType NETLIST)
                      (interface
                                (port Z (direction OUTPUT))
```

```
                                        (port A (direction INPUT))
    )))
```

A close inspection of an external list reveils that:

1 Each cell contains the list of ports as found in (any) one of its solar instances.

2 cell allows to establish an additional level of hierarchy by which a component library is hidden into an external list and each instance makes reference to the appropriate cell into a target technology libarary.

6.5 Applications

We summarize the application of the translators on different examples. The examples used in the VHDL translations are:

GCD, the Greatest Common Divisor, *BUB*, the Bubble Sorting, *FPU*, the Fixed Point Arithmetic operations $(+, -, *, /)$, *PID*, a Propotional Integral derivative, a hierarchical design instantiating the already translated FPUs, *ANSWER*, a controller for a telephone answering machine, *Motion_Estimator*, a motion estimator which is part of a video CODEC, *Motor_control*, an adaptative controller of motors for the robatic arms and *ATM*, an Asynchronous Transfer Mode.

Table 6.2 compares the VHDL translation of the above examples in terms of their lines of code before (in Solar) and after (in VHDL) the translations.

Example	nb. lines				Personalization
	Controller		Datapath		
	Solar	VHDL	Solar	VHDL	
GCD	120	189	221	398	47
Bubble	444	484	445	676	65
FPU	460	461	423	623	65
PID	719	634	427	650	63
Motion	2323	2246	490	933	282
Speed	1985	1432	717	941	38

Table 6.2: Number of Code lines before and after translation

Table 6.3 compares the EDIF translation of a sample datapath example in terms of their lines of code before (in Solar) and after (in EDIF and VHDL) the translations.

The last three columns of *Table* 6.3 indicate that only about 5% of the total number of instances belong to the Glue logic. These indicate the significance of datapath decomposition.

Example	nb. lines						nb. instances		
	DP Size = 8		DP Size = 4		Glue Logic		DP8	DP4	Glue
	Solar	EDIF	Solar	EDIF	Solar	VHDL			
Ex-1	942	821	663	600	98	150	51	35	4
Ex-2	777	671	-	-	43	60	42	-	2

Table 6.3: Number of translated decomposed datapath Code lines and Instances

We give in Appendix-8, different C++ methods for the translations of Solar into VHDL and EDIF languages together with a brief overview of the use of Programmable Translator Architecture (PAT).

Chapter 7

CONCLUSION

In this thesis work we realize a flexible, efficient and parameterized link between a High Level Synthesis and Register Transfer Level (RTL) Synthesis in the existing Computer Aided Design (CAD) environments.

- *Flexibility is achieved using a method called as the Personalization. This method offers the designers with an ability to customize the RTL architecture using a personalization file. The principal objectives are to achieve architecture synchronization and to be able to mix synthesizable and non synthesizable parts in a behavioral specification.*
- *Efficiency is attempted through a method of decomposition of the synthesized datapath at the register transfer level into two functional parts: one or more regular datapaths for datapath compilation and a glue logic for logic synthesis.*
- *Parameterization of the architecture is achieved using component generalizations in the datapath. This aspect is part of a Personalization Architecture Translator (PAT), that performs a Solar to VHDL translation.*

The above methods are currently being used in the High Level Synthesis of academic as well as some real life industrial examples with AMICAL being the High Level Synthesis system.

.

7.1 Objectives Realized

The aim is to provide a flexible, efficient and parameterized link to the RTL synthesis starting from a high level specification. The architecture is made flexible using a scheme called a *Personalization* that permits the user to engineer the architecture. An efficient architecture is aimed through a *datapath decomposition* method the purpose of which is to extract regular datapaths from the architecture available after a HLS. Finally, the parameterized link is achieved by allowing the use of a library of *generic* components.

7.2 Unsynthesizable Descriptions

Large designs, as one comes across in the Industry, need not be completely expressed for a HLS. A typical example is a specification consisting of both a control dominated and a data dominated parts. First, the two parts have to be identified and partitioned. The data dominated sub-specification would be synthesized by a system such as the Cathedral and the synthesized result would be included to the control dominated sub-specification for validation and synthesis by a system such as Amical. Such an example is presented in [BKV+96]. We have established the procedure to allow such (mixed) descriptions at a High level with the Amical system. In fact, the unsynthesized descriptions are simply migrated to the RT level.

7.3 Customized Synchronization

Synchronization is a protocol allowing the co working of the controller and the datapath in the resulting HLS architecture. Synchronization is usually achieved using a single or multiple periodic (clock) input signals. Using the Personalization method, we allow the users to choose a particular scheme of synchronization desired.

7.4 Regular Datapaths Extraction

The usual standard cell methodology for RTL synthesis doesn't take into account of the regular nature (i.e the bit-slice property) of the RT components which can be laid out using a bit-sliced layout style. However, if the RT components have different bit-widths, a large portion of the layout portion would be wasted in the bit-sliced stlye. We have presented a datapath decomposition methodology, by which, regular datapaths suitable for the datapath compilers are extracted out of the architecture.

7.5 Data Translations

HLS results available from the Amical system is an Object oriented data structure in a format called Solar. In order to support the standard language interface to the RTL synthesis, we realized two translators, namely: Solar to VHDL and Solar to EDIF translators.

7.6 Perspectives

The intermediate format Solar supports easier introduction and the extension of several methods for a procedural interface. The procedural interface facilitates to establish future methods for linking different objectives. For example, import for RTL in a different language (such as Verilog), or to achieve FPGA synthesis and so on. Another issue is the verification of HLS results. Today, we use the simulation method, but it is not exhaustive. Technique such as the Observable Time Windows [BAK92] for studying the simulation results can be incorporated. Yet another issue is to find a way to back annotate the primary information of the speed, area and power consumption details from RTL to high level since the latter is performed in a technology independent manner.

Bibliography

[AA92] T. Abbassi and S. Ben Atallah. VENUS: Modélisation multi-niveaux de systèmes VLSI dans le langage VHDL. Under graduate projet report, July 1992.

[ABOR90] R. Airiau, J.M. Bergé, V. Olive, and J. Rouillard. *VHDL Du langage à la modélisation.* Presses Polytechniques et Universités Romandes et CNET-ENST, 1990.

[ACJ96] M. Abid, A. Changuel, and A.A. Jerraya. Exploration of Hardware/Software Design Space through a Codesign of Robot Arm Controller. In *Proceedings of the European Design Automation Conference*, Geneva, Switzerland, September 1996.

[ADOJ93] M. Aichouchi, H. Ding, K. O'Brien, and A.A. Jerraya. Generation and validation of detailed architectures from behavioral VHDL descriptions. In *5th ICM (International Conference on Micrœlectronics)*, Saudi Arabia, December 1993.

[Aic94] Mohamed Aichouchi. *Study of linking architectural synthesis with RTL synthesis.* PhD thesis, TIMA-INPG, Grenoble, France, June 1994.

[AK94] S. Amellal and B. Kaminska. Functional Synthesis of Digital Systems with TASS. *IEEE trans. on CAD*, 13(5):537–552, May 1994.

[AKDJ94] M. Aichouchi, P. Kission, H. Ding, and A.A. Jerraya. Lien entre la Synthèse Architecturale et la Synthèse au Niveau Transfert de Registres. *TSI (Technique et Science Informatiques)*, 1994.

[AOJ94] M. Aichouchi, K. O'Brien, and A.A. Jerraya. Generation and Validation of detailed Architectures from behavioral VHDL Descriptions. *AJSE (Arabian Journal for Sciences and Engineering)*, October 1994.

[AVDJ94] M. Aichouchi, P. VijayRaghavan, H. Ding, and A.A. Jerraya. Linking High-Level Synthesis And Register Transfert Level tools Using VHDL. *Revue Internationale Algérienne des Technologies Avancées*, 1994.

[BAK92] R. Bergamaschi and et.al A. Kuchlmann. A Methodology For Production Use of High-Level Synthesis. In *Intl. High-Level Synthesis Wkshop*, number 6th, November 1992.

[Bak93] Louis Baker. *VHDL Programming With Advanced Topics.* WILEY, 1993.

[BCM+88] R.K. Brayton, R. Camposano, G. De Micheli, R.H.J.M. Otton, and J. Van Eijndhoven. *The Yorktown Silicon Compiler System*, pages 204–310. Ed. D.D. Gajski Addison-Wesley Publishing Company, 1988.

[BCP91] R.A. Bergamaschi, R. Camposano, and M. Payer. Datapath Synthesis using path analysis. In *Proceedings of the Design Automation Conference*, 1991.

[BD95] J.D. Sterling Babcock and A. Dollas. A Case of Study of System Synthesis with Non-Synthesizable Components using Extended VHDL. In *Proc. of Sixth IEEE International Workshop on Rapid System Prototyping*, Chapel Hill, USA, June 1995.

[BDWM93] J.R. Burch, D. Dill, E. Wolf, and G.D. Micheli. Modeling hierarchical combinational circuits. Technical report, Stanford University, Palo Alto, CA, USA, April 1993.

[BKV+96] E. Berrebi, P. Kission, S. Vernalde, S. De Troch, J.C. Herluison, J. Frehel, A.A. Jerraya, and I. Bolsens. Combined Control-flow dominated and Data-flow dominated High Livel Synthesis. In *Proceedings of the Design Automation Conference*, Las Vegas, USA, June 1996.

[BL90] J. Bhasker and H.C. Lee. An Optimizer for Hardware Synthesis. *IEEE Design and Test of Computers*, pages 20–36, 1990.

[BMK92] J. Biesenack and et.al M. Koster. The Siemens High-Level Synthesis System, CALLAS. In *Intl. High-Level Synthesis Wkshop*, November 1992.

[BNW92] J. Biesenack and et.al N. Wehl. Data Path Optimization In The CALLAS Synthesis Environment. In *Workshop on Control Dominated Synthesis From an RTL Rescription*, Grenoble, France, September 1992.

[Cam90] R. Camposano. From Behavior to Structure: High-Level Synthesis. *IEEE Design and Test of Computers*, pages 8–19, October 1990.

[Cam91] R. Camposano. Path-Based Scheduling for Synthesis. *IEEE trans. on CAD*, 10(1):85–93, January 1991.

[Cas89] A.Z. Casavant. A Synthesis Environment for Designing DSP Systems. *IEEE Design and Test of Computers*, pages 35–43, 1989.

[CB90] R. Camposano and R.A. Bergamaschi. Synthesis Using Path-Based Scheduling: Algorithms And Exercises. In *Proceedings of the Design Automation Conference*, pages 450–455, June 1990.

[CBH+91] R. Camposano, R.A. Bergamaschi, C. E. Haynes, M. payer, and S. M. Wu. *High Level VLSI Synthesis*, chapter The IBM High-Level Synthesis System. Kluwer Academic Publishers, 1991.

[Col95] B. Cole. Embedded Systems - Part 2 Digital Signal Processing. *Electronic Engineering*, page 46, April 1995.

[Cou94] B. Courtois. CAD and Testing of ICs and Systems. *Journal of Microelectronic Systems Integration*, 2(3), March 1994.

[CPTR89] C. Chu, M. Pontkonjak, M. Thaler, and J. Rabaey. HYPER: An Interactive Synthesis Environment for High Performance Real Time Applications. In *Proceeding ICCD'89*, pages 432–435, Massachusetts, 1989.

[CR89] R. Camposano and W. Rosenstiel. Synthesizing Circuits From Behavioral Descriptions. *IEEE trans. on CAD*, 8(2):171–180, February 1989.

[CRJ96] A. Changuel, R. Rolland, and A.A. Jerraya. Design of an Adaptive Motors Controller Based on Fuzzy Logic Using Behavioral Synthesis. In *Proceedings of the European Design Automation Conference*, Geneva, Switzerland, September 1996.

[CST91] R. Camposano, L.F. Saunders, and R.M. Tabet. VHDL as Input for High-Level Synthesis. *IEEE Design and Test of Computers*, pages 43–49, March 1991.

[CT89] R. Camposano and R.M. Tablet. Design Representation for the Synthesis of behavioural VHDL models. In *Proceedings of CHDL*, May 1989.

[CT90] R.J. Cloutier and D.E. Thomas. The Combination of Scheduling, Allocation, and Mapping in a Single Algorithm. In *Proceedings of the Design Automation Conference*, 1990.

[CW91] R. Camposano and W. Wolf. *High-Level VLSI Synthesis*. Kluwer Academic Publishers, 1991.

[Din96] H. Ding. *Synthèse architecturale interactive et flexible*. PhD thesis, TIMA-INPG, Grenoble, France, April 1996.

[DN89] Srinivas Devadas and A. R. Newton. Algorithms For Hardware Allocation In Data Path Synthesis. *IEEE trans. on CAD*, 8(7), July 1989.

[dPDL+89] M. Crates de Poulet, C. Duff, R. Leveugle, F. Poirot, G. Saucier, and P. Sicard. ASYL: A Logic And Architecture Design Automation System. In *Proceedings of EURO-ASIC*, pages 183–209, Grenoble, January 1989.

[DPST81] S.W. Director, A.C. Parker, D.P. Siewiorek, and D.E. Thomas. A Design Methodology and Computer Aids for Digital VLSI Systems. *IEEE trans. on Circuits System*, CAS-28(7), July 1981.

[DS91] P.J. Drenth and C. Strolenberg. Datapath Layout generation with in-the-cell routing and optimal column resequencing. In *Proceedings of the EuroDAC/EuroVHDL Conference*, Stockholm, Sweden, September 1991.

[Dut88] N.D. Dutt. GENUS: A Generic Component Library for High Level Synthesis. Technical report tr 88-22, University of California, Irvine, CA, USA, 1988.

[FF95] W. B. Frakes and C. J. Fox. Sixteen Questions about Software Reuse. *Communications of the ACM*, 38(6):75–87, June 1995.

[FI94] W. B. Frakes and S. Isoda. Success Factors and Systematic Reuse. *IEEE trans. on Software*, September 1994.

[FY69] T.D. Friedman and S.C. Yang. Methods Used in an Automatic Design Generator (ALERT). *IEEE trans. on Computers*, C-18:593–614, 1969.

[Gaj87] D. Gajski. *Silicon Compilation.* Addison-Westley, 1987.

[GBK85] E.F. Girczyc, R.J.A. Buhr, and J.P. Knight. Applicability of a Subset of Ada as an Algorithmic Hardware Description Language for Graph-Based Hardware compilation. *IEEE trans. on CAD*, pages 134–142, April 1985.

[GC93] E. Girczyc and S. Carlson. Increasing Design Quality and Engineering Productivity through Design Reuse. In *Proceedings of the Design Automation Conference*, 1993.

[GDWL92] D. Gajski, N. Dutt, A. Wu, and Y. Lin. *High-Level Synthesis : Introduction to Chip and System Design.* Kluwer Academic Publishers, Boston, Massachusetts, 1992.

[GE92] C. H. Gebotys and M. I. Elmasry. *Optimal VLSI Architectural Synthesis: Area, Performance, Testability.* Kluwer Academic Publishers, 1992.

[Gen91] GenRad:. *System HILO 4.0 System Reference Manual*, 1991.

[GKP85] J. Granacki, D. Knapp, and A.C. Parker. The ADAM Advanced Design Automation System: Overview, Planner and Natural Language Interface. In *Proceedings of the Design Automation Conference*, June 1985.

[GM92] R. K. Gupta and G. De Micheli. System level Synthesis using Reprogrammable Components. In *Proceedings of the European Conference on Design Automation*, 1992.

[GPR94] Lisa Guerra, Miodrag Potkonjak, and Jan Rabaey. System-Level Design Guidance Using Algorithm Properties. In *Proceedings of Int. Workshop on VLSI Signal Processing VII*, pages 73–82, San Diego, October 1994.

[GR94] Daniel D. Gajski and L. Ramachandran. Introduction to High-Level Synthesis. *IEEE Design and Test of Computers*, pages 44–54, Winter 1994.

[GVNG94] D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems.* Prentice Hall, USA, 1994.

[GW92] D.D Gajski and W. Wolf. *High-Level Synthesis.* Kluwer Academic Publishers, 1992.

[Har77] R. Hartenstein. *Fundamentals of Structured Hardware Design.* North Holland, 1977.

[HCLH90] C. Huang, Y. Chen, Y. Lin, and Y. Hsu. Data Path Allocation Based on Bipartite Weighted Matching. In *Proceedings of the Design Automation Conference*, pages 499–504, Orlando, 1990.

[Hil85] P.N Hilfinger. A High Level Language and Silicon Compiler for Digital Signal Processing. *IEEE Custom Integrated Circuits Conference*, pages 213–216, 1985.

[HT83] C. Y. Hitchcock and D. E. Thomas. A Method Of Automatic Data Path Synthesis. In *Proceedings of the Design Automation Conference*, 1983.

[Jac96] H. Jacobs. Silicon Technology: Risks, Oppurtunities and Challenges. In *Proceedings of the European Design and Test Conference*, Paris, France, March 1996.

[Jam86] R. Jamier. *"Génération automatique de parties opératives de circuit VLSI de type microprocesseur.* Thèse inpg, TIMA-INPG, Grenoble, France, November 1986.

[JBD+95] A.A. Jerraya, E. Berrebi, H. Ding, P. Kission, M. Rahmouni, and P. Vijayaraghavan. A pragmatic approach to behavioural synthesis. *EE - Evaluation Engineering Magazine*, May 1995.

[JD93] P. K. Jha and N. D. Dutt. Rapid Estimation for Parameterized Components in High-Level Synthesis. *IEEE trans. on VLSI*, 1(3), September 1993.

[Jer89] A.A. Jerraya. *Contribution à la Compilation de Silicium et au Compilateur SYCO.* Thèse d'état, TIMA-INPG, Grenoble, France, December 1989.

[JJ85] R. Jamier and A.A. Jerraya. APOLLON: a data-path silicon compiler. *IEEE Circuits and Device*, May 1985.

[JMGC88] A.A. Jerraya, N. Mhaya, J.P. Geronimi, and B. Courtois. SYCO - a Silicon Compiler for VLSI ASICs Specified by Algorithms. *Computer-Aided Enginneering Journal*, pages 122–130, 1988.

[JO92] A.A. Jerraya and K. O'Brien. SOLAR: An Intermediate Format for System-Level Design and Specification. In *IFIP Inter. Workshop on Hardware/software Co-Design*, Grassau, Germany, May 1992.

[JOPC92] A. Jerraya, K. O'Brien, I. Park, and B. Courtois. Towards System-Level Modeling And Synthesis. In *VLSI DESIGN'92*, India, February 1992.

[JPO93] A.A. Jerraya, I. Park, and K. O'Brien. AMICAL: An Interactive High Level Synthesis Environment. In *Proceedings of the European Conference on Design Automation*, Paris, France, February 1993.

[KCBJ93] P. Kission, E. Closse, L. Bergher, and A.A. Jerraya. Industrial Experimentation of High-Level Synthesis. In *Proceedings of the Euro-DAC/EuroVHDL Conference*, Germany, September 1993.

[KDB+95] P. Kission, H. Ding, E. Berrebi, M. Rahmouni, P. Vijayaraghavan, and A.A. Jerraya. AMICAL - High Level Synthesis system. *Journal of the Brazilian Computer Society*, November 1995.

[KDJ94] P. Kission, H. Ding, and A. A. Jerraya. Structured Design Methodology For High-Level Design. In *Proceedings of the Design Automation Conference*, San Diego, USA., June 1994.

[KDJ95] P. Kission, H. Ding, and A.A. Jerraya. VHDL based design methodology for hierarchy and component re-use at the behavioral level. In *Proceedings of the European Design Automation Conference*, 1995.

[Kis96] Polen Kission. *Exploitation de la hiérarchie et de la ré-utilisation de blocs existants par la synthèse de haut niveau.* PhD thesis, TIMA-INPG, Grenoble, France, January 1996.

[KLMM95] D. Knapp, T. Ly, D. MacMillen, and R. Miller. Behavioral Synthesis Methodology for HDL-Based Specification and Validation. In *Proceedings of the Design Automation Conference*, pages 286–291, June 1995.

[KNRR88] H. Krämer, M. Neher, G. Rietsche, and W. Rosenstiel. Data Path and Control Synthesis in the CADDY System. In *International Workshop at INPG*, Grenoble, 1988. INPG.

[Las92] R. Lasocki. PAT : Programmable Architecture Translator. Internal report, TIMA/INPG, September 1992.

[Lau92] U. Lauther. *Introduction to Synthesis: The synthesis Approach to Digital System Design.* Kluwer Academic Publishers, 1992.

[LCGM93] D. Lanneer, M. Cornero, G. Goossens, and H. De Man. An assignment technique for incompletely specified data-paths. In *Proceedings of the European Conference on Design Automation*, Paris, February 1993.

[LG88] J.S. Lis and D.D. Gajski. Synthesis from VHDL. In *Proceedings of the International Conference on Computer-Aided Design*, pages 378–381, October 1988.

[LG89] Joseph S. Lis and Daniel D. Gajski. VHDL Synthesis Using Structered Modelling. In *Proceedings of the Design Automation Conference*, 1989.

[LMWV91] P.E.R. Lippens, J.L.V. Meerbergen, A. V. Werf, and W.F.J. Verhaegh. PHIDEO, A silicon Compiler for High Speed Algorithms. In *Proceedings of the European Conference on Design Automation*, pages 436–441, Amsterdam, March 1991.

[LPCJ95] C. Liem, P. Paulin, M. Cornero, and A. Jerraya. Industrial Experiments Using Rule-driven Retargetable Code Generation for Multimedia Applications. In *Intl. Sym. Sys. Synthesis*, pages 60–65, September 1995.

[LR93] P. E. Landman and J. M. Rabaey. Power Estimation for High Level Synthesis. In *Proceedings of the European Conference on Design Automation*, pages 361–366. Paris, Feb 1993.

[Mar79] P. Marwedel. The MIMOLA design system: Detailed description of the software system. In *Proceedings of the Design Automation Conference*, pages 59–63, 1979.

[Mar85] E.R. Marx. EDIF: The Standard for Workstation Intercommunication. *IEEE Micro*, (10):68–75, October 1985.

[Mar86] P. Marwedel. A New Synthesis Algorithm For The MIMOLA Software System. In *Proceedings of the Design Automation Conference*, 1986.

[Mar91] F. Martinolle. Fusion of VHDL processes. Technical report, Stanford University, 1991.

[MBD92] J-P. Moreau, J. Borel, and D.Samani. European Trends in Library Development. *IEEE Micro*, (8):43–53, August 1992.

[MC80] C. A. Mead and L. A. Conway. *Introduction to VLSI Sytems*. Addison-Wesley Publishing Company, 1980.

[MCG+90] H. De Man, F. Catthoor, G. Goossens, J. Van Meerbergen, S. Note, and J. Huisken. Architecture-Driven Synthesis Techniques for VLSI Implementation of DSP Algorithms. *Proc. IEEE*, 78(2):319–355, February 1990.

[MD94] J. Monteiro and Srinivas Devadas. A Methodology for Efficient Estimation of Switching Activity in Sequential Circuits. In *Proceedings of the Design Automation Conference*, 1994.

[MFT+96] H. Mecha, M. Fernandez, F. Tirado, J. Septien, D. Mozos, and K. Olcoz. A Method for Area Estimation of Data-Path in High Level Synthesis. *IEEE trans. on CAD*, 15(2):258–265, February 1996.

[Mic94] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* Mc Graw Hill, 1994.

[MK88] G. De Micheli and D.C. Ku. HERCULES - a system for high-level synthesis. In *Proceedings of the Design Automation Conference*, 1988.

[MLD92] P. Michel, U. Lauther, and P. Duzy. *The Synthesis Approach to Digitial System Design.* Kluwer Academic Publishers, 1992.

[MMM95] H. Mili, F. Mili, and A. Mili. Re-using Software Issues and Research directions. *IEEE trans. on Software*, 21(6), June 95.

[MPC88] M.C. McFarland, A.C. Parker, and R. Camposano. Tutorial on High-Level Synthesis. In *Proceedings of the Design Automation Conference*, pages 330–336, 1988.

[MPC90] M.C. McFarland, A.C. Parker, and R. Camposano. The High-Level Synthesis of Digital Systems. *IEEE*, 78(2):301–318, February 1990.

[MSDP93] E. Martin, O. Sentieys, H. Dubois, and J.L. Philippe. GAUT: An architectural synthesis tool for dedicated signal processors. In *Proceedings of the EuroDAC/EuroVHDL Conference*, Germany, September 1993.

[NBD92] V. Nagasamy, N. Berry, and C. Dangelo. Specification, Planning, and Synthesis in a VHDL Design Environment. *IEEE Design and Test of Computers*, pages 58–68, June 1992.

[NSM89] J. Nestor, B. Soudan, and Z. Mayet. MIES: A Microarchitecture Design Tool. In *Proceedings of the Design Automation Conference*, 1989.

[NWM93] J. R. Nicol, C. T. Wilkes, and F. A. Manola. Object Orientation in Heterogeneous Distributed Computing Systems. *Computer*, June 1993.

[O'B93] K. O'Brien. *Compilation de silicium: du circuit au système.* PhD thesis, TIMA-INPG, Grenoble, March 1993.

[OG86] A. Orailogo and D.D Gajski. Flow Graph Representation. In *Proceedings of the Design Automation Conference*, June 1986.

[OM96] K. O'Brien and S. Maginot. Towards Maximising the use of VHDL for Synthesis. In *Proceedings of the European Design Automation Conference*, Geneva, SW, September 1996.

[OPJC93] K. O'Brien, I. Park, A.A. Jerraya, and B. Courtois. *Synthesis for Control-flow-dominated machines.* in Application-Driven Architectural Synthesis (ed.) F. Catthoor and L. Stevenson. Kluwer Academic Publishers, Boston MA USA, 1993.

[ORJ92] K. O'Brien, M. Rahmouni, and A.A. Jerraya. A VHDL-based scheduling algorithm for control-flow dominated design. In *Intl. High-Level Synthesis Wkshop*, number 6th, November 1992.

[ORJ93] K. O'Brien, M. Rahmouni, and A.A. Jerraya. DLS: A scheduling algorithm for high-level synthesis in VHDL. In *Proceedings of the European Conference on Design Automation*, Paris, France, February 1993.

[OW96] Douglas E. Ott and Thomas J. Wilderotter. *A Designer's Guid to VHDL SYNTHESIS*. Kluwer Academic Publishers, 1996.

[Pan88] B.M. Pangrle. SPLICER: A Heuristic Approach to Connectivity Binding. In *Proceedings of the Design Automation Conference*, 1988.

[Par92] I. Park. *AMICAL: Un assistant pour la synthèse et l'exploration architecturale des circuits de commande*. Thèse inpg, TIMA-INPG, Grenoble, July 1992.

[Pen86] Z. Peng. Synthesis of VLSI systems with the CAMAD design aid. In *Proceedings of the Design Automation Conference*, 1986.

[PFMH95] P. Paulin, J. Frehel, and et.al M. Harrand. High-Level Synthesis and Codesign Methods: An Application to a Videophone Codec. In *Proceedings of the European Design Automation Conference*, Brighton, UK, September 1995.

[Pis96] R. Pistorius. ArPet, An Architecture Personalization Tool. Internal report, TIMA/INPG, September 1996.

[PK89] P.G. Paulin and J.P. Knight. Force-directed scheduling for the behavioral synthesis of asics. *IEEE trans. on CAD*, 8(6):661–679, June 1989.

[PK90] C. A. Papachristou and H. Konuk. A Linear Driven Scheduling and Allocation Method Followed By An Interconnect Optimization Algorithm. In *Proceedings of the Design Automation Conference*, 1990.

[PKG86] P.G. Paulin, J.P. Knight, and E.F. Girczyc. HAL: a multi-paradigm approach to automatic data path synthesis. In *Proceedings of the Design Automation Conference*, 1986.

[POJ92] I. Park, K. O'Brien, and A.A. Jerraya. An Interactive Data-Path Allocation Algorithm. In *Workshop on Control Dominated Synthesis From an RTL Rescription*, Grenoble, France, September 1992.

[PPM+96] L. Pirmez, A. Pedroza, A. Mesquita, M. Rahmouni, P. Kission, and A. Jerraya. Analysis of different protocol description styles in VHDL for High-Level Synthesis. In *Proceedings of the European Design Automation Conference*, Geneva, September 1996.

[PS94] S. Parameswaran and M.F. Schulz. Computer-aided selection of components for technology-independent specifications. *IEEE trans. on CAD*, 13(11):1333–1350, November 1994.

[PVD95] R. Pistorius, P. Vijayaraghavan, and H. Ding. Grammar of a SOLAR file. Internal report, TIMA/INPG, September 1995.

[PVKJ96] R. Pistorius, P. Vijayaraghavan, P. Kission, and A.A. Jerraya. Personalization of the architecture produced by high level synthesis for the rt level. In *(Appearing) IFIP TC10, WG 10.5 JESSI AC-8 Workshop on Logic and Architecture Synthesis*, Grenoble, France, December 1996.

[Ram92] F.J. Ramming. *The synthesis Approach to Digital System Design*, chapter Synthesis Related Aspects of Simulation. Kluwer Academic Publishers, 1992.

[RB88] et.al. R.L. Blackburn. CORAL II: Linking Behavior and Structure in an IC Design System. In *Proceedings of the Design Automation Conference*, 1988.

[RD92] Central Research and Development. DATAMORPHOSIS : Datapath Compiler and Library. Technical report and user's guide, SGS Thomson Microelectronics, Agrate, Italy, 1992.

[RDS94] B. Rouzeyre, D. Dupont, and G. Sagnes. Component Selection, Scheduling and Control Schemes for High Level Synthesis. In *Proceedings of the European Design and Test Conference*, Paris, France, 1994.

[RES89] B. Rouzeyre, T. Ezzedine, and G. Sagnes. Operators Allocation in the Silicon Compiler SCOOP. *Integration*, pages 99–109, 1989.

[RG93a] Jan M. Rabaey and Lisa M. Guerra. Exploring The Architecture and Algorithmic Space for Signal Processing Applications. In *Technical Digest*, pages 315–319, Taejon, Korea, November 1993. Int'l Conference on VLSI and CAD.

[RG93b] E.A. Rundensteiner and D. Gajski. Component synthesis from functional descriptions. *IEEE trans. on CAD*, 12(9):1287–1299, September 1993.

[RJ95] M. Rahmouni and A.A. Jerraya. PPS: A Pipeline Path based Scheduler. In *Proceedings of the European Conference on Design Automation*, Paris, France, March 1995.

[RJ96] M. Rahmouni and A.A. Jerraya. A Taxonomy of Scheduling Algorithms for Control Flow Dominated Specifications. *IEEE trans. on CAD*, 1996. Submitted.

[RMVGG88] J. Rabaey, H. De Man, J. Vanhoof, and et.al G. Goossens. *Silicon Compilation*, chapter CATHEDRAL-II: A Synthesis System for Multiprocessor DSP Systems, pages 311–360. Ed. D.D. Gajski Addison-Westley Publishing Company, 1988.

[ROJ94] M. Rahmouni, K. O'Brien, and A.A. Jerraya. A Loop-based Scheduling Algorithm for Hardware Description Languages. *Parallel Processing Letters*, 4(3):351–364, 1994.

[RP91] V.K. Rai and C.S. Patwardhan. Automated data path synthesis to avoid global interconnects. In *Porceedings. IEEE*, pages 11–16, 1991.

[Sau87] L.F. Saunders. The IBM VHDL Design System. In *Proceedings of the Design Automation Conference*, pages 484–490, 1987.

[Sch85] Jean-Pierre Schoellkopf. *SILICIEL: Contributions a l'architecture des Circuits Integres et a la Compilation du Silicium.* PhD thesis, USMG-INPG, Grenoble, France, April 1985.

[Seq83] C. H. Sequin. Managing VLSI complexity: an outlook. *Proceedings of the IEEE*, 71(1), January 1983.

[Seq86] C. H. Sequin. VLSI design stategies. In *Proceedings of the summer school on VLSI tools and applications.* Kluwer Academic Publisher, July 1986.

[SIA94] The National Technology Roadmap for SemiConductors. Technical report, SemiConductor Industry Association, San Jose, California, 1994.

[SJKR91] C.B. Shung, R. Jain, and et.al. K. Riemey. An Integrated CAD System for Algorithm-Specific IC Design. *IEEE trans. on CAD*, 10(4):447–463, April 1991.

[SKDB96] L. Stok, D.S. Kung, and et.al. D. Brand. BooleDozer: Logic Synthesis for ASICs. *IBM Journal of R&D*, 40(4), 1996.

[SL93] C. Safinia and R. Leveugle. *Synthesis for Control Dominated Circuits*, chapter Clocking scheme selection for circuits made up of a controller and a datapath. IFIP Ed. G. Saucier and J. Trilhe, 1993.

[SMG95] S. Swamy, A. Molin, and B. Govnot. OO-VHDL: Object-oriented extensions to VHDL. *Computer*, October 1995.

[SN91] et.al S. Note. CATHEDRAL III: Architecture-driven High-Level Synthesis for High Throughput DSP Applications. In *Proceedings of the Design Automation Conference*, 1991.

[Sou83] J.R. Southard. MacPitts: An Approach to silicon compilation. *Computer*, 16(12), December 1983.

[Sta94] W. Staringer. Constructing Applications from Reusable Components. *IEEE trans. on Software*, September 1994.

[Sto91] L. Stok. *Architectural Synthesis and Optimization of Digital Systems.* PhD thesis, Eindhoven University of Technology, 1991.

[Syn92] Synopsys:. *Synopsys Design Analyzer Reference Manual, version 3.0*, December 1992.

[Syn93] Synopsys:. *Synopsys VHDL System Simulator Tutorial, Version 3.0b*, June 1993.

[Syn94] Synopsys:. *Synopsys Behavioral Compiler User Guide, Version 3.2a*, October 1994.

[TDW+88] D.E. Thomas, E.M. Dirkes, R.A. Walker, J.V. Rajan, J.A. Nestor, and R.L. Blackburn. The system Architect's Workbench. In *Proceedings of the Design Automation Conference*, pages 337–343, June 1988.

[Tho83] D.E. Thomas. Automatic data path synthesis. *Computer*, December 1983.

[Tie88] R.A. Tierney. Modelling complex systems. *VLSI System Design*, May 1988.

[TM91] D. E. Thomas and P. Moorby. *The VERILOG Hardware Description Language*. Kluwer Academic Publishers, 1991.

[Tri87] H. Trickey. Flamel: A high-level hardware compiler. *IEEE trans. on CAD*, pages 259–269, 1987.

[TRLG81] S. Trimberger, J. A. Rowson, C. R. Lang, and J. P. Gray. A Structered Design Methodology and Associated Software Tools. *IECS*, 28(7), July 1981.

[TS83] C. Tseng and D. S. Siewiorek. Facet: A procedure for the automated synthesis of digital system. In *Proceedings of the Design Automation Conference*, 1983.

[TWRT88] C.J. Tseng, R.S. Wei, S.G. Rothweiler, and M.M. Tong. Bridge: A Versatile Behavioral Synthesis System. In *Proceedings of the Design Automation Conference*, pages 415–420, 1988.

[VCV+95] C.A. Valderrama, A. Changuel, P. Vijayaraghavan, M. Abid, T.B. Ismail, and A.A. Jerraya. Hardware Software co-design based on unified C/VHDL models. In *Proceedings of the European Design and Test Conference*, Paris, France, March 1995.

[VG88] N. VanderZanden and D. Gajski. MILO: a MIcro architecture and Logic Optimizer. In *Proceedings of the Design Automation Conference*, 1988.

[VHD87] IEEE, NY. USA. *IEEE Standard VHDL Langage Reference Manual*, March 1987.

[Vij93] P. Vijayaraghavan. Linking High Level Synthesis into an existing CAD environment. Master's thesis, TIMA-INPG, Grenoble, France, September 1993.

[Vij94a] P. Vijayaraghavan. Architecture decomposition. In *5th JESSI AC-8 Workshop on Synthesis, Optimization*, Munich, Germany, March 1994.

[Vij94b] P. Vijayaraghavan. Architecture personalization. In *6th JESSI AC-8 Workshop on Synthesis, Optimization*, Munich, Germany, September 1994.

[Vij94c] P. Vijayaraghavan. How to write a personalization file. Internal report, TIMA/INPG, December 1994.

[VRB+93] J. Vanhoof, K. V. Rompaey, I. Bolsens, G. Goossens, and H. De Man. *High-Level Synthesis for Real-Time Digitial Signal Processing*. Kluwer Academic Publishers, 1993.

[Wan88] E. Wang. A compiler for silage. Master's thesis, University of California, Berkeley, USA, December 1988.

[WC91] R.A. Walker and R. Camposano. *A Survey of High-Level Synthesis Systems*. Kluwer Academic Publishers, Boston MA USA, 1991.

[WC95] Robert A. Walker and S. Chaudhuri. Introduction to the scheduling problem. *IEEE Design and Test of Computers*, pages 60–68, Winter 1995.

[Wis89] Adrian Wise. *Introduction To Motion Picture Coding and the CCITT Algorithm*, December 1989.

[WM92] R. Woudsma and J.L. Van Meerbergen. A Driving Force for High-Level Synthesis of Signal-Processing Architectures. *IEEE Micro*, (8):20–33, August 1992.

[Wol91] W. Wolf. *High Level VHDL Synthesis*, chapter Architectural Optimization Methods For Control Dominated Machines. Kluwer Academic Publishers, 1991.

[WS89] N. S. Woo and H. C. Shin. A technology-adaptive allocation of functional units and connections. In *Proceedings of the Design Automation Conference*, 1989.

[WT87] R.A. Walker and D. E. Thomas. Design representation and transformation in the system architect's workbench. In *Proceedings of the International Conference on Computer-Aided Design*, pages 166–169, 1987.

[WTH+92] W. Wolf, A. Takach, C. Y. Huang, R. Manno, and E. Wu. The princeton university behavioral synthesis system. In *Proceedings of the Design Automation Conference*, 1992.

[Zim79] G. Zimmermann. The MIMOLA Design System: A Computer Aided Digital Processor Design Method. In *Proceedings of the Design Automation Conference*, 1979.

.

Appendix-1:
Typical Components Used by AMICAL

Cell Name	*Port Name*	*Port Direction*	*Port Type*	*Port Property*
ADD, ALU, SUB	Sel	IN	Array	Unresolved
	in1	IN	Array	Resolved
	in2	IN	Array	Resolved
	out1	OUT	Array	Resolved
FU_AS, ADDSUB	clk	IN	Bit	Unresolved
	Sel	IN	Bit	Unresolved
	com	IN	Array	Unresolved
	in1	IN	Array	Resolved
	in2	IN	Array	Resolved
	out1	OUT	Array	Resolved
FU_shift	com	IN	Array	Unresolved, CONTROL
	in1	IN	Array	Resolved, DATA_NONGENERIC
	in2	IN	Array	Resolved, DATA_NONGENERIC
	out1	OUT	Array	Resolved, DATA_NONGENERIC

Table .1: Functional Units

Cell Name	*Port Name*	*Port Direction*	*Port Type*	*Port Property*
Switch	Sel	IN	Bit	Unresolved, Control
	S_IN	IN	Array	Resolved
	S_OUT	OUT	Array	Resolved
Mux_2	Sel	IN	Bit	Resolved, Control
	in1	IN	Array	Resolved
	in2	IN	Array	Resolved
	out1	OUT	Array	Resolved

Table .2: Switching Units

Cell Name	*Port Name*	*Port Direction*	*Port Type*	*Port Property*
ConstREG	Data_OUT	OUT	Array	Resolved
FU_ram	Sel	IN	Array	Unresolved
	indata	IN	Array	Resolved
	inadr	IN	Array	Resolved
	outdata	OUT	Array	Resolved
FU_rom	Sel	IN	Bit	Unresolved
	com	IN	Array	Unresolved, CONTROL
	out1	OUT	Array	Resolved, DATA_NONGENERIC
Flag_REG	reset	IN	Bit	Unresolved
	clk	IN	Bit	Unresolved
	W	IN	Bit	Unresolved, Control
	Data_IN	IN	Array	Resolved
	Data_OUT	IN	Array	Resolved
	CR	OUT	Array	Resolved
VariableRegister	reset	IN	Bit	Unresolved
	clk	IN	Bit	Unresolved
	W	IN	Bit	Unresolved, CONTROL
	Data_IN	IN	Array	Resolved
	Data_OUT	OUT	Array	Resolved

Table .3: Storage Units

Cell Name	*Port Name*	*Port Direction*	*Port Type*	*Port Property*
ExtCON	E_IN	IN	Array	Resolved
	E_OUT	OUT	Array	Resolved

Table .4: External Communication Unit

Cell Name	*Port Name*	*Port Direction*	*Port Type*	*Port Property*
pc_po_int_cell,	Clk	IN	Bit	Unresolved
po_pc_int_cell	data_in	IN	Bit	Unresolved
	data_out	OUT	Bit	Unresolved

Table .5: The Interface Scan Cells

Appendix-2:
Algorithmic Description used in Ch-5

```
ENTITY bub_clean IS
  PORT( reset : IN  BIT;
        clk   : IN  BIT;
        start : IN  BIT;
        ackin : OUT BIT;
        validin   : IN  BIT;
        datain: IN  INTEGER;
        ackout: IN  bIT;
        validout  : OUT BIT;
        dataout: OUT INTEGER);
END bub_clean;

ARCHITECTURE behavior OF bub_clean IS
TYPE memory IS ARRAY (0 TO 3) OF INTEGER;

FUNCTION decr2(in1:IN INTEGER) RETURN  INTEGER IS
 VARIABLE res1: INTEGER;
BEGIN
  res1 := in1 - 2;
  RETURN res1;
END decr2;

FUNCTION get255 RETURN INTEGER IS
BEGIN
  RETURN 3;
END get255 ;

BEGIN

Bubblesort :PROCESS
  VARIABLE i, k, iter, size    : INTEGER;
  VARIABLE t1, t2, t3, t4      : INTEGER;
  VARIABLE ram                 : memory;

PROCEDURE fillram IS
BEGIN
   i := 0;
   size := get255;
   WHILE (i <= size) LOOP
         WAIT UNTIL (validin = '1');
         t1 := datain;
         ackin <= '1';
         WAIT UNTIL (validin = '0');
```

```
         ram(i) := t1;
         ackin <= '0';
         i := i + 1;
   END LOOP;
END fillram;

PROCEDURE emptyram IS
BEGIN
   i := 0;
   WHILE (i <= size) LOOP
         IF (ackout /= '0') THEN WAIT UNTIL (ackout = '0'); END IF ;
         t1 := ram(i);
         dataout <= t1;
         validout <='1';
         WAIT UNTIL (ackout = '1');
         validout <= '0';
         i := i + 1;
   END LOOP;
END emptyram;

BEGIN
   IF (start /= '1') THEN WAIT UNTIL (start = '1'); END IF ;
   fillram;
   i := 1;
   WHILE i <= size LOOP
         iter := size + 1;
         WHILE iter > i LOOP
            t1 := iter - 1;
            t2 := decr2(iter);
            t3 := ram(t1);
            t4 := ram(t2);
            iter := t1;
            IF t3 < t4 THEN
                ram(t1) := t4;
                ram(t2) := t3;
            END IF;
        END LOOP;
        i := i + 1;
   END LOOP;
   emptyram;
END PROCESS Bubblesort;

END behavior;
```

The keyword signal is mandatory and is followed by the name of the signal, *signal_name*. This signal can be of single BIT type or of type Array. In the latter case, the *signal_name* is of the form (array *identifier_V signal_size*) where all three parts of the signal name are mandatory and the signal size *signal_size* is a positive integer. The specified signal (BIT or ARRAY) would be added to the corresponding block of *identifier_U*. The personalization action also depends on the following *signal_qualifiers*.

- signal_qualifiers ::= (**dir** *legal_value_for_DIR*)
  (**attr** *legal_value_for_ATTRIBUTE*) (**type** *identifier_D*) *list_of_locals*
- list_of_locals ::= **empty** || (**local** *identifier_V*)
- legal_value_for_DIR ::= $IN||OUT||INOUT||INTERNAL$
- legal_value_for_ATTRIBUTE ::= $CLOCK||RESET||others$

The first *signal_qualifier* defines the direction dir of the signal in the block (DesignUnit) and is mandatory. The direction is given by the *legal_value_for_DIR* and might be either IN, OUT, INOUT or INTERNAL. If the direction is IN, OUT or INOUT, we can define the first personalization action, the Port–Personalization. In case of a direction INTERNAL, we define the second personalization action, the Net–Personalization.
Furthermore, we may need to personalize the *Glue* cells before executing the Net–Personalization, because the glue cells appear as Instances in the modules and their ports may be connected to other ports by nets. This will be explained in the third personalization action, the Glue–Personalization. These three actions are directly applied on the contents of the personalization file. Finally, we will introduce other personalization actions that result as a consequence of the above introduced ones.

4.4.1 Action–1: Port–Personalization

A signal *signal_name* with a direction dir in the form of *legal_value_for_DIR* (IN, OUT or INOUT) becomes a port in the personalized circuit. The attribute attr of the signal called *legal_value_for_ATTRIBUTE* might either be CLOCK, RESET, DATA, CONTROL or any other string value meeting the syntax of the personalization file. Furthermore, the signal attribute is mandatory only in the case of the CLOCK and the RESET signals for the CONTROL block of the circuit, in order to add the necessary synchronization information in the form equivalent of a VHDL synchronization process. In a SOLAR description, this CONTROL block is identified by its ViewType *"behavior"*.
The signal type type is mandatory for all ports with the direction IN, OUT or INOUT and it might be any *identifier_D* that can be a legal VHDL Data Type of a port.

Finally, the *signal_qualifier*, namely, the *list_of_locals* is optional in the case of the Port–Personalization.

In *Table* 4.1, the signal specifications in a personalization file and their equivalences as ports in the personalized SOLAR file (DesignUnit) are summarized.

signal	**PORT**	**requirement**
signal_name:		
1. *identifier_V*[101]	*identifier_V*	mandatory
2. **Array** *identifier_V signal_size*[102]	**Array** *identifier_V signal_size*	mandatory
signal_qualifier dir:		
dir	**DIRECTION**	mandatory
legal_value_for_DIR[103]	*legal_value_for_DIR*	mandatory
signal_qualifier attr:		
attr	**PROPERTY PortType**	mandatory[104]
legal_value_for_ATTRIBUTE	*legal_value_for_ATTRIBUTE*	mandatory
signal_qualifier type:		
type	**PROPERTY Type**	mandatory
identifier_D[105]	*identifier_D* (of PROPERTY Type)	mandatory
In the case of a type **std_logic_vector** (IEEE) or **BusX** (MVL7)	**PROPERTY Resolved BusX**	—
signal_qualifier local:		
local	no personalization action required	optional
identifier_V	no personalization action required	optional

Table 4.1: *Port* Personalization

The following example explains the action of the Port–Personalization. In this example we will consider the personalization of the SOLAR file **test_example.solar**, to which an input port **test_port** has to be added. The corresponding personalization file contains:

```
(global example.global
   (block test_example
          (signal test_port (dir IN)
```

[101] *signal_name* with direction IN, OUT or INOUT in the 1–BIT case; might be any identifier string *identifier_V* that can be a legal name for a VHDL port or signal

[102] *signal_name* with direction IN, OUT or INOUT in the case of an Array; the size might be any integer number greater than zero

[103] with direction IN, OUT or INOUT

[104] only for the CLOCK and the RESET port obligatory in order to add the required synchronization process necessary for the translation into VHDL format

[105] might be any *identifier_D* that can be a legal VHDL Data Type of a port

```
                                (attr Anything)
                                (type STD_ULOGIC)
                                (local Nothing))
          ( ... )
)  )
```

The resulting action is an addition of a port to the interface of the design unit named test_example in the test_example.solar file as follows:

```
(PORT test_port (DIRECTION IN) (BIT)
                (PROPERTY PortType Anything)
                (PROPERTY Type STD_ULOGIC))
```

4.4.2 Action–2: Net–Personalization:

A signal *signal_name* with a direction dir in the form of *legal_value_for_DIR* INTERNAL, becomes a net in the personalized circuit. The attribute attr of the signal called *legal_value_for_ATTRIBUTE* can be CLOCK, RESET, DATA, CONTROL or any other string value meeting the syntax of the personalization file. Furthermore, the signal attribute is absolutly optional in the case of the direction INTERNAL.
The signal type type is also optional for all ports with the direction INTERNAL and it is determined by the ports to which the net will be connected.
Finally, the *signal_qualifier list_of_locals* is mandatory in the case of the Net–Personalization. This *signal_qualifier* determines the name of the ports, which will be connected by the internal signal specified in the personalization file. This Personalization step will traverse all the ports of the design and the ports within the Instances and connect them through the net corresponding to the *global* signal.

In *Table* 4.2, the signal specifications in a personalization file and their equivalences as nets in the personalized SOLAR file are summarized.

The following example explains the Net–Personalization action. In this example we consider the personalization of a SOLAR DesignUnit contained in a file **module_example.solar**, wherein an INTERNAL–port named connect_net has to added. The corresponding personalization file contains:

```
(global example.global
```

[201] *signal_name* with direction INTERNAL in the 1–BIT case; might be any identifier string *identifier_V* that can be a legal name for a VHDL port or signal

[202] *signal_name* with direction INTERNAL in the case of an Array; the size might be any integer number greater than zero

[203] with direction INTERNAL

[204] as only a direction INTERNAL will be personalized as a net, the equivalence of *dir* and *JOINED* expresses a direction INTERNAL

[205] might be any identifier string *identifier_V* that can be a legal name for a VHDL port or signal

[206] if the *identifier_V* caracterizes the port of an instance

signal	NET	requirement
signal_name: 1. *identifier_V*[201] 2. **Array** *identifier_V signal_size*[202]	 Net_*identifier_V* Net_*identifier_V*	 mandatory mandatory
signal_qualifier dir: **dir** *legal_value_for_DIR*[203]	 **JOINED** __[204]	 mandatory mandatory
signal_qualifier attr: **attr** *legal_value_for_ATTRIBUTE*	 no personalization action required no personalization action required	 optional optional
signal_qualifier type: **type** *identifier_D*	 no personalization action required no personalization action required	 optional optional
signal_qualifier local: **local** *identifier_V*[205] —	 **PORTREF** *identifier_V* **INSTANCEREF**[206]	 mandatory mandatory mandatory[206]

Table 4.2: *Net* Personalization

```
   (block module_example
       (signal connect_net (dir INTERNAL)
                           (attr Anything)
                           (type STD_ULOGIC)
                           (local all_ports))
       ( ... )
)  )
```

Assuming, that a port named *all_ports* exists in the block *this_block* of the DesignUnit **module_example.solar** and that this DesignUnit posseses an input with the same name, *all_ports*, the following net will be added to the list of the nets of the DesignUnit **module_example.solar**:

```
(NET Net_connect_net
     (JOINED
        (PORTREF all_ports  (INSTANCEREF this_block))
        (PORTREF all_ports)
)    )
```

4.4.3 Action–3: Glue–Personalization

A Glue cell *glue* with the name identifier_UV, becomes an instance (block) having the same name within the personalized circuit. The name of the glue cell identifier_UV is mandatory and must correspond to the name of the SOLAR file (omitting the standard extension .solar). Personalization looks for the SOLAR description of this glue cell, either in the current working directory or in a specified SOLAR library directory pointed to by a shell variable called *AMICAL_PAT_libDIR*. Once the Personalization action detects the required file, this file is read and the ports of the glue cell would be linked to the respective *portinstances* of the instance (block) that has been added in the personalized design.

In *Table* 4.3, the glue specifications in a personalization file and the corresponding SOLAR DesignUnit together with their equivalences as an instance and portinstances in the personalized SOLAR file are summarized.

glue (Personalization)	**INSTANCE**	**requirement**
glue_name:		
identifier_UV[301]	*identifier_UV*	mandatory
VIEW (Solar File)	**VIEWREF**	**requirement**
view_specification	*view_specification*	mandatory
VIEWTYPE ¨*view_type*¨	*name_of_glue_cell_solar_node*	mandatory
Interface part:		
INTERFACE	no personalization action required	optional
list_of_ports		
PORT	**PORTINSTANCE**	mandatory
port_name	*port_name*	mandatory
signal_qualifier for direction		
DIRECTION	no personalization action required	optional
legal_value_for_dir	no personalization action required	optional
signal_qualifier for signal width		
BIT	no personalization action required	optional
—	**PROPERTY PortType GLUE**	optional[302]

Table 4.3: *Glue* Personalization

In the following example, we consider the personalization of a SOLAR designunit contained in a file **another_example.solar**, wherein only the GLUE–cell **this_glue** has to be added. The corresponding personalization file contains:

```
(global example.global
```

[301] *identifier_UV* means any identifier string that satisfies UNIX file name conventions (with the exception of wild cards: ∗, ?) or that can be a legal name for a VHDL port/signal

[302] in order to caracterize the instance source

```
   (block another_example
      (glue this_glue)
      ( ... )
)  )
```

Assuming, that a SOLAR DesignUnit contained in a corresponding file name **this_glue.solar** exists either in the working directory or in the specified SOLAR library directory, this glue cell will be added in the list of the instances of the DesignUnit **another_example**.

Given a following SOLAR DesignUnit of the form:

```
(SOLAR this_one
   (DESIGNUNIT This_one
      (VIEW implementation (VIEWTYPE "communicating")
          (INTERFACE
            (PORT one_port (DIRECTION IN) (BIT))
            ( ... )
)  )  )   )
```

the following instance will be added to the DesignUnit *another_example*:

```
(INSTANCE This_one
    (VIEWREF implementation this_one
       (PORTINSTANCE  one_port (PROPERTY PortType GLUE))
)   )
```

Appendix-6 contains a figure showing the effect of Glue–Personalization on a simple example. The figure shows how the IN–Ports of the instance **Clock** have been connected to the IN–Ports of the DesignUnit *gcd_circuit*.

Hereunder, the personalization rules that are indirectly invoked by the personalization of an architecture are introduced. These rules combine the personalization using the personalization file and the personalization actions enhanced by the SOLAR (component) library.

A design module in the form of a Solar file may contain Instances representing DesignUnits, that are all to be personalized by using the same personalization file. Therefore, we must be able to update only the ports of those Instances by using this personalization file using the next personalization action, called as the Instance – Personalization. Furthermore, these DesignUnits are to be added as components to the architecture, i. e. the Personalization searches them in the SOLAR library and adds them to the circuit. Finally, when all updates are made, the netlist must be updated in order to correctly connect together the ports and portinstances.

4.4.4 Action–4: Instance–Personalization

While personalizing a circuit (the top module), it may contain instances that are to be personalized by using the same personalization file. The personalization information for the instances can be found in the list of blocks containing one block for the circuit and other blocks for the instances within this circuit. In order to add the synchronization nets to the complete circuit, the instances must contain the ports that should be added via the personalization file, in form of portinstances.
Thus, a signal *signal_name* with a direction dir in the form of *legal_value_for_DIR* (IN, OUT or INOUT) becomes a portinstance in the personalized instance of the circuit. The *signal_qualifier list_of_locals* are optional in the case of the Instance–Personalization.

In *Table* 4.4, the signal specifications in a personalization file and their equivalences as ports in the personalized SOLAR DesignUnit are summarized.

signal	**PORTINSTANCE**	**requirement**
signal_name:		
1. *identifier_V*[401]	*identifier_V*	mandatory
2. **Array** *identifier_V signal_size*[402]	**Array** *identifier_V signal_size*	mandatory
signal_qualifier dir:		
dir	—[403]	mandatory
legal_value_for_DIR[404]	—	mandatory[404]
signal_qualifier attr:		
attr	**PROPERTY PortType**	optional
legal_value_for_ATTRIBUTE	*legal_value_for_ATTRIBUTE*	optional
signal_qualifier type:		
type	no personalization action required	optional
identifier_D	no personalization action required	optional
signal_qualifier local:		
local	no personalization action required	optional
identifier_V	no personalization action required	optional

Table 4.4: *Instance* Personalization

In the following example we consider the personalization of a SOLAR file **last_example.solar**, in which only the *input* port **last_test_port** has to be added as a portinstance to the

[401] *signal_name* with direction IN, OUT or INOUT in the 1–BIT case; might be any identifier string *identifier_V* that can be a legal name for a VHDL port or signal

[402] *signal_name* with direction IN, OUT or INOUT in the case of an Array; the size might be any integer number greater than zero

[403] mandatory, because only signals with direction IN, OUT or INOUT will be added as portinstances

[404] with direction IN, OUT or INOUT

instance **this_block**. The personalization file contains:

```
(global example.global
   (block this_block
      (signal last_test_port (dir IN)          (attr  Anything)
                             (type STD_ULOGIC) (local Nothing))
      ( ... )
)  )
```

In the list of the instances of the DesignUnit **last_example.solar** , the following portinstance will be added to the instance **this_block**:

```
(PORTINSTANCE  last_test_port (PROPERTY PortType Anything))
```

4.4.5 Action–5: Component Adding

In this step the DesignUnits corresponding to the instances of the architecture are added in form of DesignUnits to the circuit. As the instances in a synthesized circuit only contain portinstances and the corresponding portreferences, this information is not sufficient for the translation into VHDL by the translator. Additional information like the VHDL type or, in case of a vector, the array width are required. They are contained in the SOLAR files in the SOLAR library, in the form of one file per each component (InstanceRef) containing the corresponding DesignUnit. Then these DesignUnits are added as components to the complete architecture.

The example below shows the component adding for the architecture **gcd_circuit**, to which the DesignUnit Clock will be added. Before the personalization, the files **gcd_circuit.solar** and the file **Clock.solar** (in the SOLAR library) are the following:

```
(SOLAR AMICAL_for_Circuit_with_interface
   (DESIGNUNIT gcd_circuit
      (VIEW AbstractArchitecture (VIEWTYPE "Structure")
          (INTERFACE
             (PORT first_in  (DIRECTION IN)  (BIT))
             (PORT second_in (DIRECTION IN)  (BIT))
             (PORT first_out (DIRECTION OUT) (BIT))
             ( ... )
)  )  )   )

(SOLAR clock
   (DESIGNUNIT Clock
      (VIEW implementation (VIEWTYPE "communicating")
         (INTERFACE
            (PORT clk      (DIRECTION IN)    (BIT))
            (PORT reset    (DIRECTION IN)    (BIT))
```

```
            (PORT cont_clk (DIRECTION OUT)   (BIT))
            (PORT cont_reset (DIRECTION OUT) (BIT))
            ( ... )
)  )  )  )
```

After personalization, both files are merged into one file of the form:

```
(SOLAR AMICAL_for_Circuit_with_interface
   (DESIGNUNIT gcd_circuit (PROPERTY DesignType ARCHITECTURE)
       (VIEW AbstractArchitecture (VIEWTYPE "Structure")
           (INTERFACE
              (PORT first_in   (DIRECTION IN) (BIT))
              (PORT second_in  (DIRECTION IN) (BIT))
              (PORT first_out  (DIRECTION OUT)(BIT))
              ( ... )
           )
       )
   )
   (DESIGNUNIT Clock (PROPERTY DesignType COMPONENT)
       (VIEW implementation (VIEWTYPE "communicating")
          (INTERFACE
             (PORT clk        (DIRECTION IN)  (BIT))
             (PORT reset      (DIRECTION IN)  (BIT))
             (PORT cont_clk   (DIRECTION OUT) (BIT))
             (PORT cont_reset (DIRECTION OUT) (BIT))
             ( ... )
)  )   )  )
```

4.4.6 Action–6: Net–Updates

The final personalization step consists in the updating of the nets. Therefore, all signals declared in the personalization files that are specified with a direction dir in the form of *legal_value_for_DIR*, which might either be IN, OUT or INOUT, are read again. If a *signal_qualifier list_of_locals* with a list of *identifier_V* is present, the corresponding port (after Port–Personalization, Action–1) is connected to all portinstances with the name *identifier_V* in all instances, wherever the portinstance appears.

4.5 Results

The architecture, after a personalization, contains the elements from the personalization file added to the abstract architecture (circuit) available from a high level

synthesis.

4.5.1 Personalized Architecture Contents

The basic parts of the architecture are a controller and a datapath.

- **Controller:** In the intermediate format *Solar*, a Controller is specified using the view Behavior; the functionality is expressed using a *StateTable* having several *States*, each of them containing the actions to be performed (usually conditional *Assign*ment of values to variables), following to which is the next state to jump to. Controller is an infinitely running finite state machine (generated by Amical is of type *Mealy*). No instance(s) shall be present within a behaviour view. The statetable is analogous to a VHDL *process* containing a *Case* statement in which, each *when* corresponds to one state of the StateTable. During the personalization, some additional port(s) may be added to the Controller's Interface, if specified in the personalization file, after ensuring the uniqueness.

- **Datapath:** Datapath in Solar is specified using the view Structure since it contains *instance*(s) of the Functional Units. The ports of the instances are connected by *net*s. Thus, the functionality of a datapath is modelled as a structural netlist interconnecting ports of the functional units. A port of a FU may either be connected to the *interface*, i.e., one of the *port*s, or to a *port* of another instance, or may be left unconnected. An unconnected port of a direction other than the *in*put may be modelled using the VHDL *open* keyword, however, if an input port is desired to be unconnected, it is necessary to introduce a dummy *signal* of the correct size (or a parameterized signal), and to connect it to the open input port. During the personalization, some additional port(s) may be added to the datapath's Interface, if specified in the personalization file. Other possible additions are instances specified as a *glue* in the personalization file, and thus the required nets are used to interconnect the glue instance(s) with the interface. This permits to shield the functionality of the glue instance, which may be already present as an existing component or a black-box for a deferred implementation.

- **Circuit:** Circuit refers to the interconnection of Controller and Datapath instances addressed above. Obviously, structure is the required view. During the personalization, some additional port(s) may be added to the circuit's interface, if specified in the personalization file as well as to connect the additional port(s) present in either of the controller or of the datapath or of both.Glue instances are added exactly in the same manner as explained above for the datapath.

4.5.2 Applications

The *personalization* is one of the key factors for the success of a HLS tool. It is used in order to support a flexible architecture at the RTL. Personalization permits to link the result of the HLS tool with an existing RTL environment containing already existing components. We use two examples in order to explain the results:
The first example called *speed* is used in the speed control of robotic motors [CRJ96, ACJ96] . The second example called *motion* is a motion estimator used with a video codec [BKV+96].
The speed control circuit is placed in between a memory and a set of motors running at diffrent speeds. Communication between these two external sides is achieved using the *I/O units*. An I/O unit is a specific functional unit that can exchange the data with the external world without the control of the speed controller. *Figure* 4.3 shows an extract of the speed control circuit realized using the high level synthesis and the personalization.

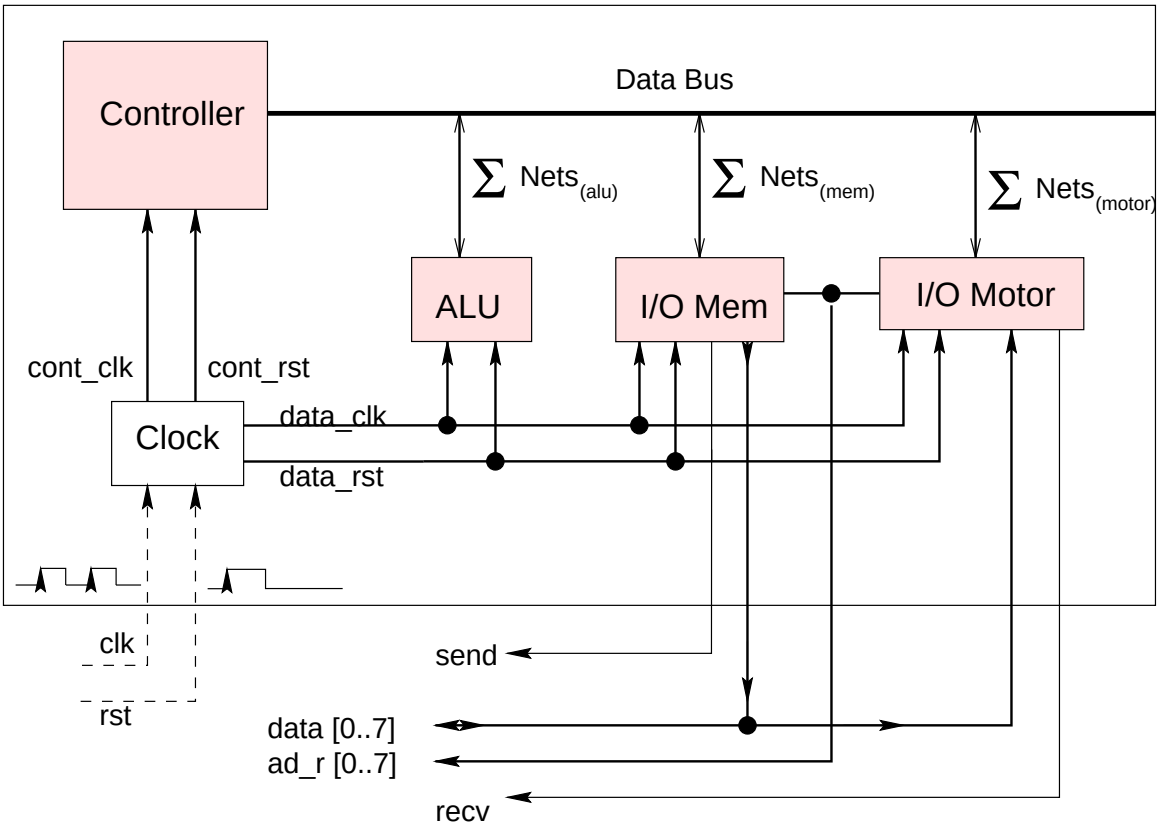


Figure 4.3: Part of the speed control circuit.

In the *Figure* 4.3, the result of AMICAL is constituted by the shaded blocks and the solid lines. The other parts have been added by the personalization step by using the personalization file *speed.global* shown below. The two interface units called as the *I/O Mem* and the *I/O Motors* are used in the speed control circuit as the simple functional units that allow to *read* and *write* the internal registers. These units exchange data

with the external world using a sophisticated protocol based on the *wait* loops and a *hand-shaking* protocol. The use of these interface units avoid the duplication of the protocol exchange procedures used in the top controller. The protocol is further sub contracted to the interface unit.

A part of the personalization file used for the *circuit* in the *speed* example is given below:
Contents required for the synchronization scheme are illustrated using *italic* letters, whereas, those needed for the migration of the unsynthesized high level description are underlined. Thus we illustrate the methods of handling a user specified synchronization scheme as well as how a part of an unsynthesized high level description is moved to the RTL synthesis.

ABBREVIATIONS:

s_u_l: Std_Ulogic; *s_u_l_v*: Std_Ulogic_Vector.

```
(global speed.global
   (block speed_circuit
      (signal clk       (dir In)       (attr Clock) (type s_u_l)   (local clk))
      (signal rst       (dir In)       (attr Reset) (type s_u_l)   (local rst))
      (signal cont_clk  (dir Internal) (attr Clock) (type s_u_l)   (local cont_clk))
      (signal cont_rst  (dir Internal) (attr Reset) (type s_u_l)   (local cont_rst))
      (signal data_clk  (dir Internal) (attr Clock) (type s_u_l)   (local data_clk))
      (signal data_rst  (dir Internal) (attr Reset) (type s_u_l)   (local data_clk))
      (signal send      (dir Out)      (type s_u_l)  (local SEND1))
      (signal recv      (dir Out)      (type s_u_l)  (local recv1))
      (signal (array data 8)           (dir InOut)   (type s_u_l_v) (local DATA1))
      (signal (array ad_r 8)           (dir Out)     (type s_u_l_v) (local Ad_R1))
      .
      .

      (glue   Clock)
   )
)
```

The *motion* estimator [BKV+96] makes use of three functional units: Two memories and a DSP co-processor. *Figure* 4.4 shows an extract of the motion estimator circuit realized using the high level synthesis followed by the personalization.

In the above circuit, the communication between the functional units and that with the external world is achieved without invoking the top controller.

1 Examples for the *inter FU communication* is shown in *Figure* 4.4 using the lines *A1* and *A1_CRAM*. These indicate that the processor uses the memories without the top controller being addressed. In fact, the functional units *Mem1* and *Mem2* are used as dual port memories after the high level synthesis and a personalization, while they were used as single port memories for the high level synthesis.

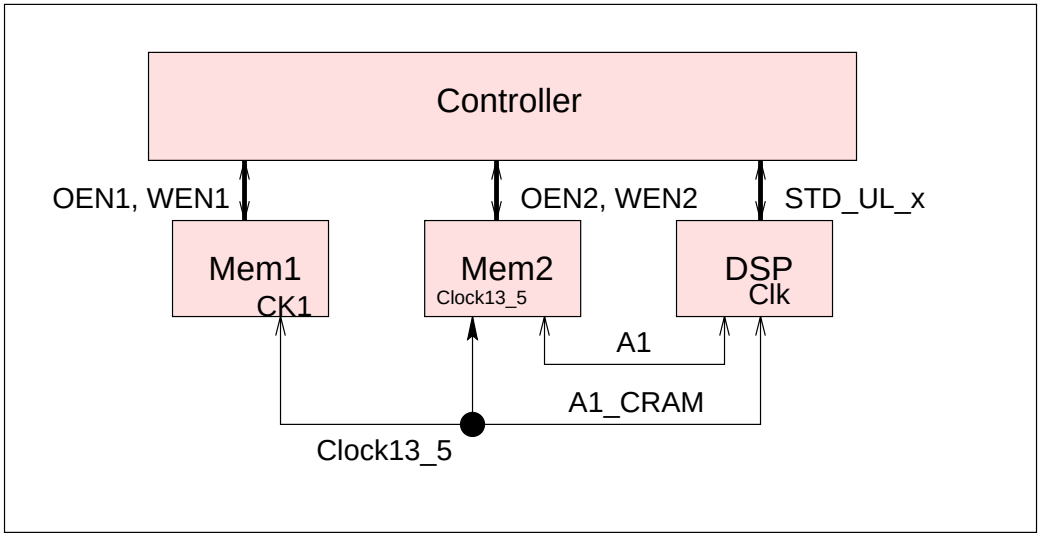


Figure 4.4: Part of the architecture of the motion estimator circuit.

2 Examples for the communication between a functional unit and the *external world* is shown in *Figure* 4.4 using the lines *CK1* and *Clock13_5* and *Clk*. Once again, the communication is possible, without addressing the top controller. This allows to have an asynchronous communication with the external world [BKV+96].

A part of the personalization file used in the *datapath* of the *motion* example is given below.
Using this example, we explain how we handle different naming conventions in binding the already existing components: for example in the first signal definition: *CK1, clock13_5, clk* are the component ports having different names that all connect to the *dp_clk13_5* signal.

```
(global motion.global
  (block motion_datapath
     (signal dp_clk13_5 (dir IN)
                        (attr clock)
                        (type std_ulogic)
                        (local CK1) (formal clock13_5) (formal clk))
     (signal (array std_ulv8_0 8)   (dir INTERNAL)
                        (attr CRAM)
                        (type std_ulogic_vector)
                        (local D2)  (formal std_ulv8_0))
     (signal std_ul_0   (dir INTERNAL)
                        (attr CRAM)
```

```
                        (type std_ulogic)
                        (local OEN2)(formal std_ul_0))
      (signal std_ul_1   (dir INTERNAL)
                        (attr CRAM)
                        (type std_ulogic)
                        (local OEN1)(formal WEN2) (formal std_ul_1))
      (signal (array A1_CRAM 8)      (dir INTERNAL)
                        (attr sel_CRAM)
                        (type std_ulogic_vector)
                        (local A1) (formal A1_CRAM))
      .
      .
   )
)
```

Table 4.5 summarizes the list of contents added using the personalization in our two demonstrative examples. The table contains the number of *signal*s and *glue* blocks added by personalization. The support for naming conventions is shown under the column *Name*, the synchronization scheme using the column *Sync* and, the mixing of un-synthesizable into synthesizable specifications using the column *Mix*.
The credits for the example *speed* goes to A.Changuel and that for the *motion* to E.Berrebi,P.Kission and J.Frehel.

Example	Controller			Datapath			Circuit			Glue Cells	
	signal			signal			signal				
	Name	Sync	Mix	Name	Sync	Mix	Name	Sync	Mix	Sync	Mix
SPEED	2	2	-	10	2	8	8	6	8	1	-
ME	-	4	5	13	3	26	1	11	15	1	1

Table 4.5: Signals and Glue cells added via Personalization

4.6 Conclusion

We have presented a method called *personalization* in this work. The objectives of this development have been to mix the synthesizable and unsynthesizable descriptions in the high level specifications and to customize the synchronization scheme required for the register transfer level (RTL) architecture available after a high level synthesis. The results are validated using simulations:

We presented, including an industrial design example, how this method makes available the RTL architecture for synthesis in an existing Computer Aided Design (CAD) environment, The method introduces a user specifiable file, called a *personalization file*, that is specified in a simple list based format. The method is part of a *Programmable Architecture Translater, PAT* used with the *Amical* HLS tool.
The ongoing and future work consists of

1 finding automatic ways to analyze and compare the simulation results at the high level and the RTL taking into account of the two different timing concepts,

2 providing a method to generate the personalization files automatically.

We give in Appendix-6, more details on the methods used to achieve the personalization action.

Chapter 5

DECOMPOSITION

In this chapter, we present a method of Datapath Decomposition. *The contribution of this method is to enable an efficient link between a high level synthesis and a datapath layout compilation. Today, High Level Synthesis enables the automatic generation of a* datapath. *However, it may not be directly used with a classical datapath layout compiler. The main reason for this difficulty is the use of datapath components of different data widths. In this chapter, we explore a* decomposition *method that partitions a datapath available from a high level synthesis into several* regular *datapaths. These regular datapaths can then be compiled by using the classical datapath layout compilers. The method is illustrated using the AMICAL high level synthesis system and an industrial datapath compiler . The application of this method offers the advantages of regularity, parameterization as well as some possible saving in the area of the layout as compared to a logic synthesis.*

.

5.1 Introduction

Datapath is invariably a part of most of the complex VLSI designs and thanks to the *high level synthesis*, datapaths can today be generated automatically. After scheduling and allocation, a high level synthesis results in an architecture composed of a controller and a datapath. *Figure* 5.1 gives the standard synthesis flow from a high level to the register transfer level and thus a datapath compilation.

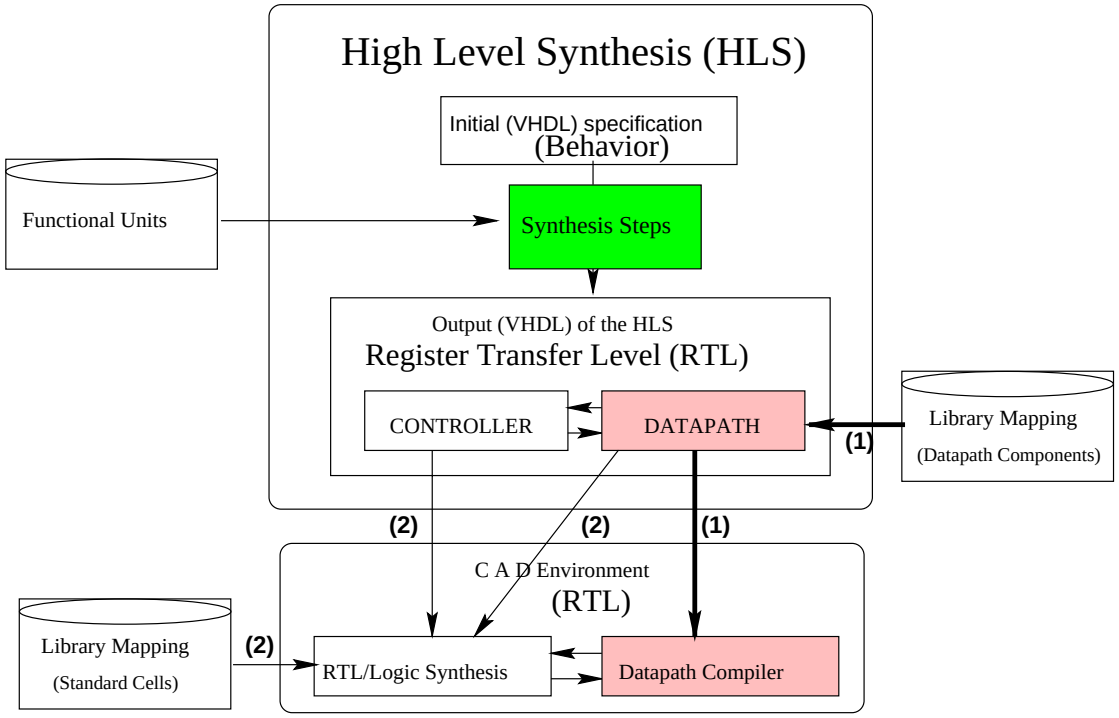


Figure 5.1: Flow of High Level Synthesis to a Datapath Compilation.

5.1.1 Background

The different steps involved in a high level synthesis (HLS) have been well studied. Today, the use of HLS is becoming more and more reliable. Experiments show that the high level synthesis permits to reduce the specication size of about 5 times on a significant industrial design [BKV+96]. The success is also due to a reliable link from the result of high level synthesis to the existing CAD environment solving the engineering problems such as the synchronization between the controller and the datapath, multiple stage pipe lining and so on . This is also aided by using the standard input-ouput languages such as the *VHDL[VHD87], Verilog, etc*. The resulting controller, in the form of a finite state machine, is ideally synthesized using any of the register transfer level synthesis tools, such as the *Synopsys , Mentor, Compass, Cadence etc*. The datapath generally includes non standard components supporting data of different widths.

5.1.2 Problem Statement

A *datapath compiler* is a tool that generates regular datapaths in an efficient way in terms of area and speed. However, the datapath compiler handles only the regular datapaths containing the components of a specified technology library and acting on a single data width. The datapath compiler generates a bit-slice structure [RD92] made up of an array of an 1-bit datapath function. High level synthesis generates a datapath made of a netlist of *Functional Unit*s. A functional unit can either be a standard component, such as a register, memory, ALU, or an already existing macro block for re-use, or a newly specified (complex) function. Due to the initial higher level abstraction, a high level synthesis may not yield a datapath containing functional units expressable in terms of the basic technology components and thus can not be directly imported by a classical datapath compiler. This problem can be addressed by defining higher level operators for the datapath functions. However, the different functional units contained within the datapath may be operating on data of different widths [1] for efficiency reasons and thus resulting in an irregular datapath. Of course, the datapath can undergo a logic synthesis if the functional units can be described in a dataflow level. This method allows a boolean optimization, but the idea of parameterizing the regular blocks such as Memory, ALU etc., would be lost as compared to a datapath compiler approach. The main reasons causing a datapath unable to be imported by a standard datapath compiler are:

1 The functional units are abstract: This can be solved by providing generic implementation details of the functional units at a structural level.

2 Functional units do not map directly onto a given datapath library: To solve this, the functional units are to be broken down into a netlist of lower level mappable components. We devise a *flattening* tool to enable this task.

3 Heterogenous data widths: A datapath compiler can not by itself handle a mix of multi widthed data ports among the different functional units. The solution consists of a certain *decomposition* steps, that can extract the regular datapaths specified by their widths.

In this chapter, we present a *decomposition* method permitting the compilation of the so obtained datapath by using a classical datapath compiler. The datapath available from a high level synthesis may contain several regular datapaths tied to a glue logic. Each regular datapath is characterized by a parameter known as the *width* W_i for expansion by a datapath compiler. The *decomposition* method described in this chapter is used to extract the regular datapaths in a format suitable for import by a datapath compiler. To demonstrate the method, we present an example in the

[1] *Width* is measured in terms of the number of bits while expressed in the binary notation.

context of the AMICAL high level synthesis system and a datapath compiler used by the SGS-Thomson Micro Electronics.

We begin by a *Formulation* of our approach in Section-5.2. In Section-5.3, we explain its *Implementation* and in Section-5.4, some of the *Results* obtained using an example. Finally, we *conclude* by discussing the merits (vs.) the overhead of this approach in Section-5.5.

5.2 Formulation

5.2.1 Regular and Ir-regular datapaths

A *Regular* datapath is one which contains components having data ports of a *single* size. It can also contain a certain number of ports to effect the control logic, however, the sizes of such control ports donot vary with the data ports. A *Non-Regular* datapath is one containing components having data ports of *different* sizes. It can contain one or several of the regular datapaths each of which can undergo a datapath compilation and also a *glue-logic* that may be subject to a logic synthesis and a boolean optimization. A datapath compiler allows to perform a *bit-slice* expansion on a regular datapath function, expressed for a single bit, into that one, for a specified number, termed as *Width*, of bits. The expansion does not alter the functionality which is a structure defined in terms of a library of components, termed as datapath library, known to the datapath compiler. Each component above must contain the *port*s (of direction = in, out, inout) for communication; The ports are either of type *data* or of *control*. The datapath compiler organizes the initial structure in the form of an array containing the *row*s as the ports of data and the *column* as the ports for the control. (or vice-versa). An n-bit (given that the width desired = n) structure is obtained by performing the bit-slice expansions on each of the rows (and thus the data ports), and using the column (and thus the control ports), only for appropriate inter-connections (Ex: connection of $Cout_j to Cin_{j+1}$).
Figure 5.2 explains the geometry of a datapath expansion while *Table* 5.1 contains certain geometrical characteristics of the datapath. It's worth noting that, all the *data* ports, entering horizontally, are subject to bit-slice expansion by the datapath compiler. On the other hand, the *control* ports enter vertically and do not undergo any expansion; This permits to have the bare minimum number of bits for the control logic part of the datapath.

5.2.2 Datapath Compiler Library

The datapath compiler makes use of a library of components for operations in order to perform the operations such as the *arithmetic, logic, memory etc.* If a datapath

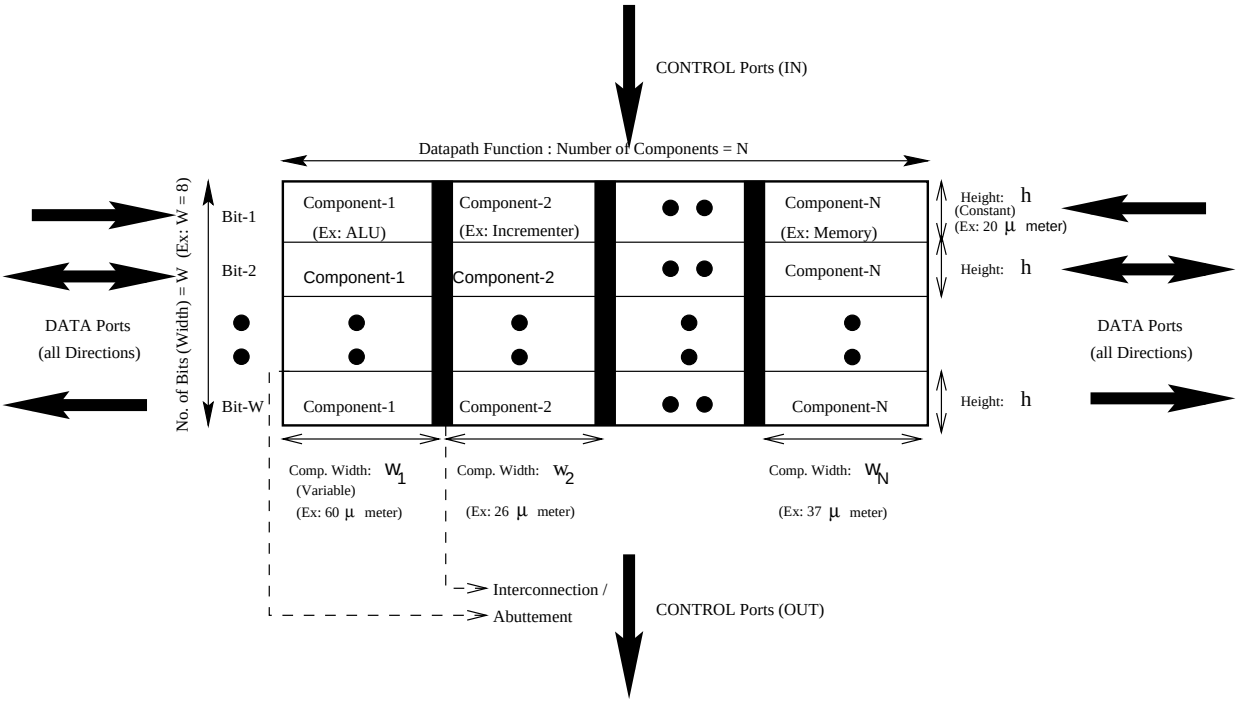


Figure 5.2: Expansion of a Regular datapath

description includes a complex component which is not a part of this library, then a *Flattener* is required in order to break up the complex component into a functionally equivalent netlist of components from the datapath compiler library. Thus, before performing a datapath decomposition, a pre-processing step of flattening is carried out.

5.2.3 The Decomposition

The datapath resulting after a high level synthesis with AMICAL contains a data-structure given in a hierarchical form as follows:

- Solar_Datapath_File_Contents ::= (**solar** identifier *designunit_description*)
- designunit_description ::= (**designunit** identifier *view_description*)
- view_description
  ::= (**view** identifier
  viewtype_description
  property_description
  interface_description
  contents_description)
- view_type_description ::= (**viewtype** identifier)

parameter	value	units
component *height* (constant)	h	μ metre
components per bit-slice	N	-
component *width* (variable)	$w_1, w_2, ...w_N$	μ metre
Datapath Width (No. of Bits)	W	-
Perimeter (minimum)	2*(W*h + $\sum_{j=1}^{N} w_j$)	μ metre
Area (minimum)	(W*h*$\sum_{j=1}^{N} w_j$)	square μ metre

Table 5.1: Geometry of the datapath

- property_description
  ::= **empty (property** *property_name property_value* **)**
- property_name ::= identifier
- property_value ::= identifier
- interface_list ::= **(interface** port_description **)**
- port_description ::= port_declaration { port_declaration }
- port_declaration
  ::= **empty (port** identifier **(direction** identifier **)** (type))
  | **(array** identifier **(direction** identifier **)** (type))

Each *port* of the datapath interface has a property called *PortType*. The port type is introduced in order to distinguish between the *control* and the *data* ports. The size of the control ports shall remain the same for the datapath compilation, whereas the data ports are parameterized as per the size of a regular datapath. In order to identify a regular datapath, a property called *width* is introduced to each port of an *instance*. A *Net* is used to connect two or more ports. Nets do not have any direction and may be divided into three groups:

N_{feed} is a *feed-through* that connects two or more interface ports.

N_{ext} is an *external* net that connects ports of 1 or more instances to the interface.

N_{int} is a *local* net which is invisible outside the datapath block. It connects atleast 2 ports of the datapath instances and shall not contain any of the interface port.

A net satifisfies the following properties:

1 *Merging*: 2 or more nets containing a common port can be merged. For example: if a Net N1 connects ports A and B, another net N2 connects ports B and C, then N1 merged to N2 gives a single net N12 connecting A, B and C.

2 *Cutting*: Any net connecting more than 2 ports can be cut into several nets: For example, if N1 is a net connecting ports A, B and C then, N1 can be cut into 2 nets as: N11 connecting A , B and N12 connecting B , C.

3 *Adding*: Any netlist containing at least one port unconnected or open allows addition of a new net. For example, if an input port of an instance is left open, a net may be added to connect it to the ground to avoid an unknown initial value on the open port.

4 *Uniqueness*: Each net is unique and also satisfies the following relation: $(N_{ext} \bigcup N_{feed}) \bigcap N_{int} = \quad \emptyset$

We recall that the *decomposition* is a process of extracting a regular datapath of a specified width out of a non-regular datapath. The datapath resulting (initial) from a high level synthesis is the first non-regular datapath. If the initial datapath contains multiple regular datapaths, then it is subject to several steps of iterative bi-partitioning. Each step results in a regular datapath and its compliment, which we term as the *glue*. The decomposition results are shown in *Table* 5.2.

Width	Ports		Instances		Nets		
(> 0)	Data	Control	Datapath	Glue	Nfeed	Next	Nint
W_1	P_{d1}	P_{c1}	I_{d1}	I_{g1}	$Nfeed_1$	$Next_1$	$Nint_1$
.	.	.	.	.	.	.	.
W_n	P_{dn}	P_{cn}	I_{dn}	I_{gn}	$Nfeed_n$	$Next_n$	$Nint_n$

Table 5.2: Datapath *decomposition* using the (x n) bi-partitions

5.3 Implementation

Figure 5.3 shows the organization of the implementation of the *datapath decomposition*. It is divided into three steps as explained in the sub-sections 5.3.1 through 5.3.3.

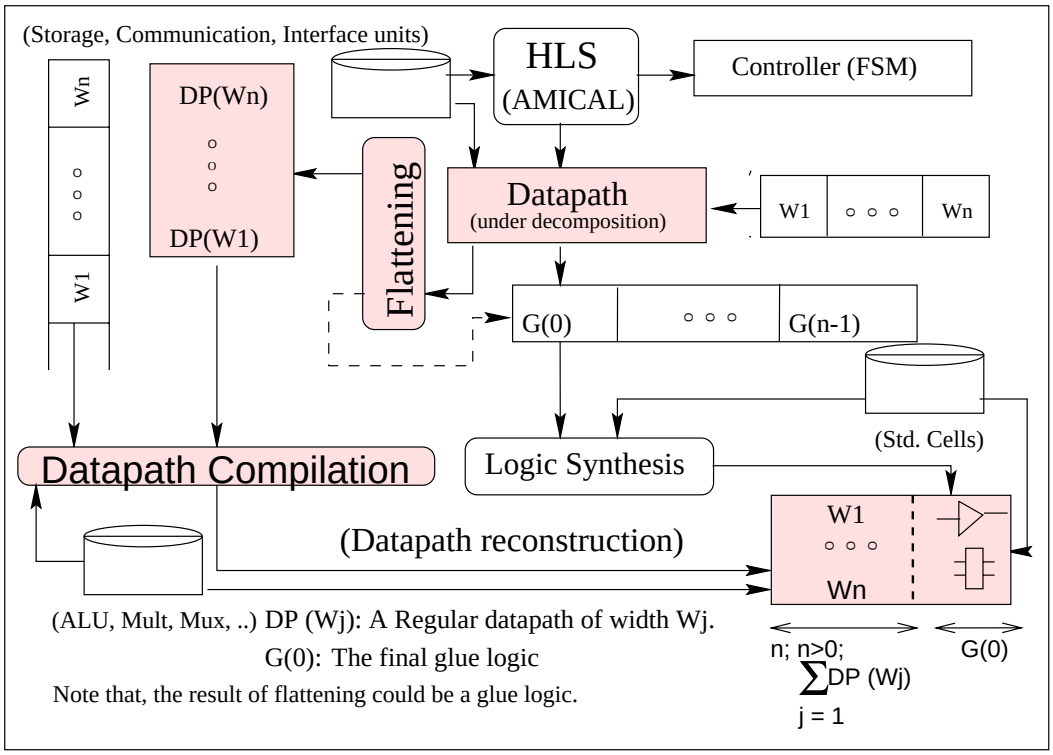


Figure 5.3: Implementation of the datapath decomposition

5.3.1 Component Flattening

The flattening process expands each instance of the datapath into its lower level implementation in terms of datapath and/or standard cell components. This process is similar to the Functional Synthesis approach presented in [AK94]. The functional synthesis approach synthesizes a functional description of one or possibly several RTL components into component(s) from a given (technology specific) library. After a flattening step, every component of the datapath and the instances may become in terms of a certain number of components from a *datapath* library and the remaining from a *glue* logic library. The latter would undergo a logic synthesis. The port properties, especially, the Port Type which is either *control* or *data* and the instance port *width* which is a positive integer that must be added to the components of the datapath libarary.
Using *Figure* 5.4, we explain a possible realization of a *register* upon component using the above libraries for the *flattening* [Vij94a].

The initial data is available in the solar format In this case, the flattening will replace each instance of *register* by a netlist of primitives as shown in the bottom of *Figure* 5.4. These primitives may correspond to the components from the datapath library that are parameterized by the width of the register. The primitives can also belong to a glue logic. The operations are performed on a object oriented data structure available by the result of compiling the datapath description in Solar. The contents

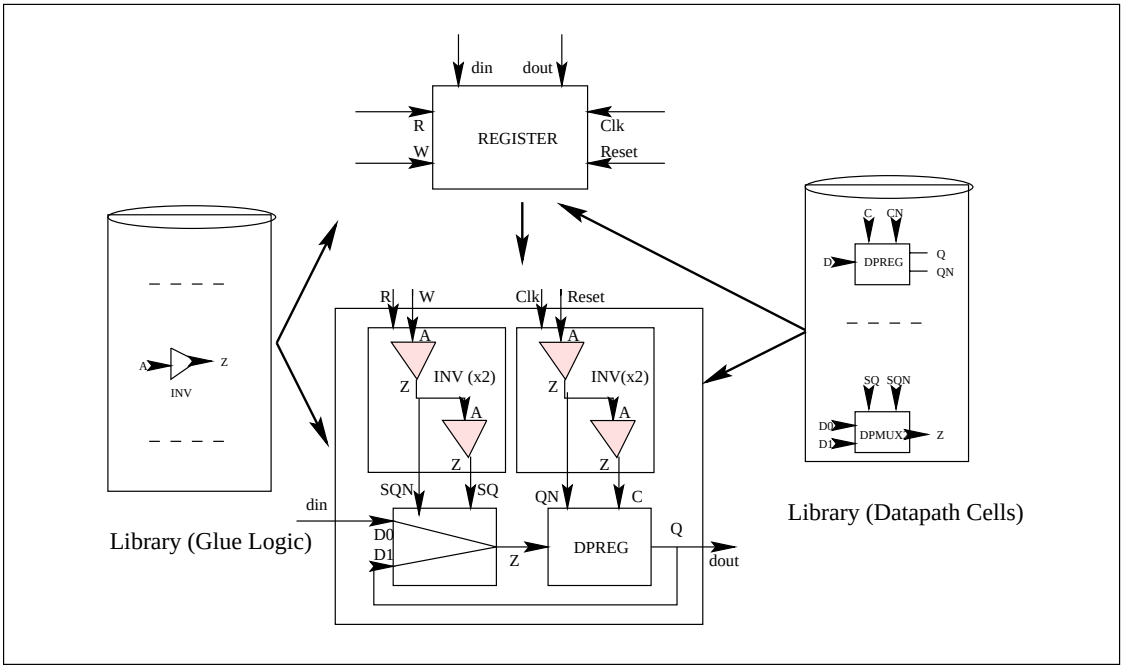


Figure 5.4: An Example: Flattening a Register for the datapath decomposition

of the datapath are managed in terms of specific *Objects*. The result of flattening is again a data structure in which:

- The Port objects are not modified w.r.t the initial contents.
- The instance objects are first expanded in terms of leaf instances that come from the libraries. The names of the leaf instances are then modified to contain the global references as explained here under:

  $\forall$ `instance` $/=$ `null`
  $n(I_{new}) = concat(n(I_{initial}) + n(I_{leaf}))$

  where,
  nI_x stands for the instance name and
  $+$ indicates the concatenation or join sign, which is typically an underscore character.

- Similarly, the net objects are flattened to compute the global port names:

```
∀ net / = null
   ∀ portref / = null
     if instref / = null
         ∀ instance / = null
```

```
            if NameIR == nI
                fetch component (instance) from the new library
                   ∀ leafNet
                      if NamePR == nP
                           n(P_new) = n(P_initial) + n(P_leaf))
                      endif
            endif
          end forall
      endif
    end forall
  end forall
```

where,
nI indicates the name of an instance ,
nIR is the name of an instanceref referred by a portref ,
leafNet is a net found in the fetched library component ,
nP indicates the name of a port ,
nPR is the name of a port of the library component fetched and
+ indicates the concatenation sign, which is typically the underscore character.

5.3.2 Regular Datapath Extraction (or) Splitting

A datapath *splitter* is defined in order to extract one or several of the regular datapath(s) contained within the result of flattening. Each regular datapath is characterized by a parameter, called *width*, that guides a *datapath compiler* for the bit-slice expansion. The splitting is introduced using *Figure* 5.5, in which:

- the *instances*
  - I_{jk} represents an instance k of I_j after the flattening and before a splitting.
  - I_{dk} represents the k-th regular datapath instance of I_j after splitting k-times.
  - I_g represents the glue logic instance of I_j after a maximum of n-splits.
- the *nets*
  - N_{feed} is a feed through net.
  - N_{ext} is a net that connects atleast one external port fron the set (P_{input}, P_{output}, P_{in-out}.
  - N_{int} is a net that does not connect to any of the external ports (as shown by the dashed lines).

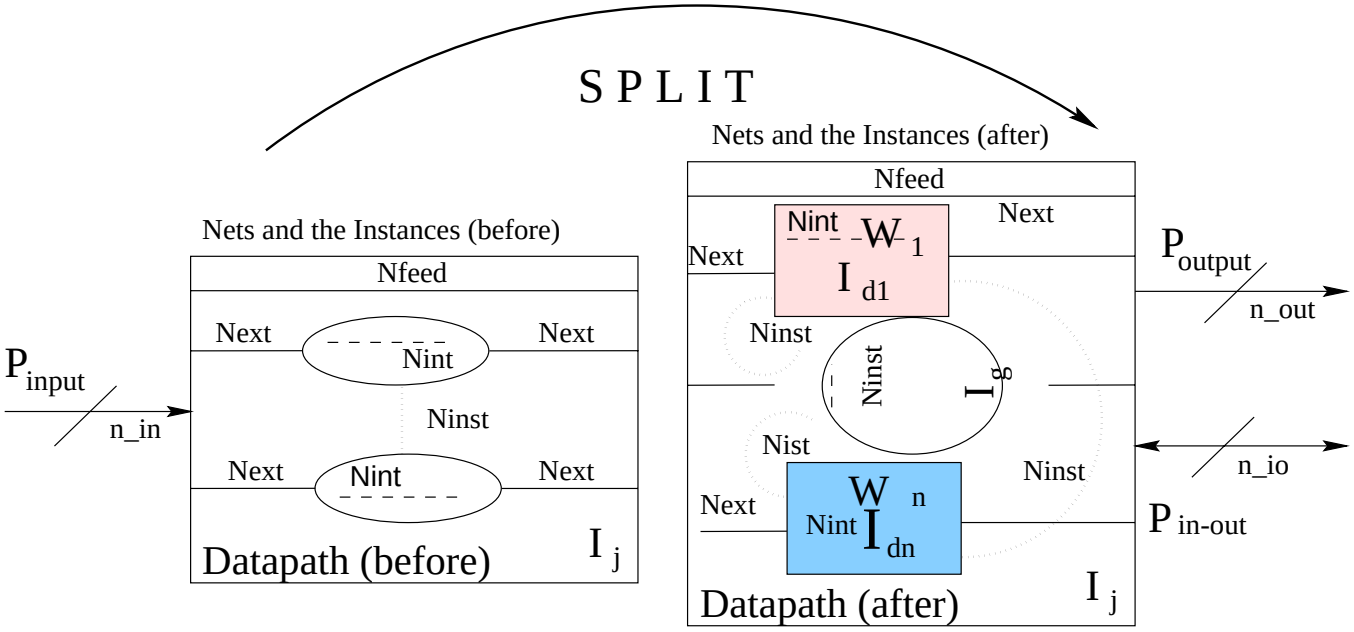


Figure 5.5: An (xN) Iterative splitter for datapath decomposition

- N_{inst} is a net that connects atleast two different instances, say, I_p and I_q (as shown by the dotted lines).

- the *ports* are not affected by splitting. They are thus shown only once.

Given a flattened datapath specification contained in a file [2] F, the splitter decomposes it into a set of one bit regular datapath(s) $F_{d1}, F_{d2}, .., F_{dn}$ and F_g where,
F_{dj} is a regular datapath of width Wj and
F_g is a glue-logic satisfying the condition:

$$F = \sum_{j=1}^{n} F_{dj} + F_g$$

The + sign indicates the addition of the functionality of the glue-logic to that of the datapath(s).
The objective of the splitter is to derive a data structure containing:

$$P_g, I_g, N_g and \sum_{j=1}^{n} P_{Wj} + I_{Wj} + N_{wj}$$

where,
the ports, instances and nets are indicated by their first letters, respecctively: *P, I,* and *N*.
The subscript g stands for the glue and W_j indicates a regular datapath of the above width. In the rest of this section, we shall follow the three terms: *dp* means a

[2] operating system (UNIX) terminology.

regular datapath, *glue* means any valid data for a logic synthesis such that: $glue \bigcap$ dp = Øand *global* means the reconstruction of the initial datapath using the dp and glue partitions and is functionally equivalent to the initial datapath. Further, we abbreviate, the dp ports as P_{dp}, the port(s) of the glue as P_{glue} and that of global as P_{global}. The procedures used to decompose the objects of *port*s, *instance*s and *net*s are given below:

1 Splitting the **PORT**s

```
∀ port / = null add to global.
  ∀ net / = null
    if (n_feed)
       No action
    else if (n_ext)
       switch (current net is connected to)
              case P_dp:  add to dp
              case P_glue:  add to g
              case P_global:  add to gl
       end switch;
    else if (n_inst)
       compute a new port for dp;
       compute a new port for g;
    else if (n_int)
       No action
    else
       Message:  current net is undefined.
    end if
  end ∀ net
end ∀ port
```

2 Splitting the **INSTANCE**s

```
∀ instance
  ∀ portinstance
    if (property(porttype) == DATA&&(property(width) == SPLIT_SIZE))
       add instance to dp; update dp instance name list
    endif
  end ∀ portinstance
  if (name of the instance ⊂ dp instance name list)
     add instance to glue; update glue instance name list
  endif
end ∀ instance
```

where,
$SPLIT_{SIZE}$ represents the *Width* of a regular datapath.

3 Splitting the **NET**s

```
∀ net
  if n_ext∈ (external ports) + (only portrefs from a dp.)
            add net to the dp;
  else if n_ext∈ (external ports) + (only portrefs from the glue.)
            add net to the glue;
  else if n_int∈ (only portrefs from one dp.)
            add net to the dp;
  else if n_int∈ (only portrefs of one glue.)
            add net to glue;
  else if n_inst∈ (only portrefs of several dps.)
            add net to the dp;
  else if n_inst∈ (only portrefs of several glues.)   [3]
            add net to glue;
  else if
            n_inst∈ (portrefs of some dps and the rest of glue.)
            compute new (local) portref(s) in the connecting dp(s);
            compute new (local) portref(s) in the connecting glue(s);
  else
            Message:  No action.
  endif
end ∀ net
```

During the computation of a new *portref*, in the case of nets of type n_{inst}, a ceratin conflicts may arise. Infact, when a net joins 2 ports of instances for merging, their identities in terms of their names, directions, types need not be homogenous. Resolution rules are required in cases of difference(s). In *Table* 5.3, we present the resolution rules that are frequently used. If a net joins more than 2 ports, the rules will be applied recursively.

In addition, the *Port Size* is determined as:
$Psize_{12} = Max(Psize_1, Psize_2)$ and
the *Port Kind* is determined to be scalar iff, $(Pkind_1 = Pkind_2 = scalar)$
else, the (Port Kind = vector). Only in the latter case Psize has a significance.

5.3.3 Datapath Reconstruction

The *global* datapath is reconstructed by the interconnection of extracted regular datapath(s) multiplied by their respective widths and the glue. It is validated by a

Port Parameter	value Port1	value Port2	case	value resolved
Port Name	Name1	Name1	$P_1(N) = P_2(N)$	Name1
	Name2	Name2	$P_2(N) = P_1(N)$	Name2
	Name1	Name2	$P_1(N) /= P_2(N)$	Name1+Name2
Port Direction	in	in	$P_1(\phi_i) = P_2(\phi_i)$	in
	out	out	$P_1(\phi_j) = P_2(\phi_j)$	out
Note: X = Any =	inout	X	$P_1(\phi_k) =$ inout	inout
in, out, inout.	X	inout	$P_2(\phi_k) =$ inout	inout
	in	out	$P_1(\phi_i) /= P_2(\phi_j)$	inout
	out	in	$P_1(\phi_j) /= P_2(\phi_i)$	inout
Port Type	control	control	$P_1(T_i) = P_2(T_i)$	control
	data	data	$P_1(T_j) = P_2(T_j)$	data
	control	data	$P_1(T_i) /= P_2(T_j)$	data
	data	control	$P_1(T_j) /= P_2(T_i)$	data

Table 5.3: *Port* resolution rules for N_{inst}

simulation to ensure the initial functionality. After a reconstruction, the internal data structure are translated into standard formats:

1 A regular *datapath* is translated into an EDIF format for an import by the datapath compiler.

2 The *glue* is translated into the VHDL format for a (data flow) logic synthesis.

3 The *global* is translated into the VHDL format for a functional validation by the simulation and then imported by the lower level synthesis tools towards the silicon.

Since the EDIF descriptons correspond to the netlists of the regular datapaths of 1-bit, the port sizes can be specified using the comments for the designer.

5.3.4 Datapath Area

After the decomposition, the *cell area* of the datapath can be obtained by the summation of three parts as shown in the *Figure* 5.6:

$A_{datapath} = A_{data} + A_{glue}$.
where,
A_{data} represents the cell area corresponding to the data part that will multiply,
A_{glue} represents the cell area corresponding to the area of the glue cells.

$$A_{data} = \sum_{j=1}^{n} (W_j * L_j)$$

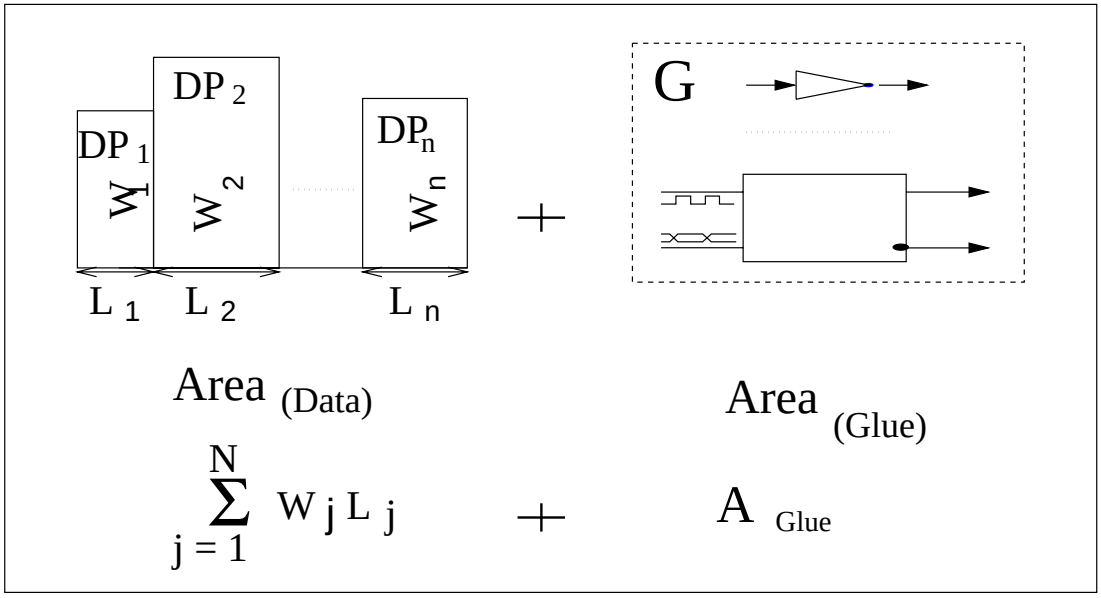


Figure 5.6: Datapath Area calculation

where,
($n >= 1$) represents the number of the regular datapaths.
W_j represents the width of the regular datapath j and
L_j represents a length quantified by the total area of the cells contained within its 1-bit function.

A_{glue} may be obtained using any of the logic syntesis tools. This calculation ignores the interconnection area which can be obtained only after a layout synthesis.

5.4 Results

Figure 5.7 shows the flow of the decomposition of a datapath containing two regular datapaths and thus undergoing the decomposition. In *Figure* 5.7, **W** denotes the abbreviation to a positive width; the process *transEDIF* translates a solar description into EDIF while the process *transVHDL* translates a solar description into VHDL. These results correspond to the design of a *bubble sort* algorithm [Aic94].

As may observed in this flow, the initial datapath is flattened. The flattened datapath is then splitted into a regular datapath of width 8 and rest of the contents (Rest). In the next step, the Rest undergoes splitting to result another regular datapath of width 4 and a final glue. The two regular datapaths would be expanded by the datapath compiler, while the glue would be synthesized using logic synthesis.

Now, we present the objects contained within this example following the *flattening*

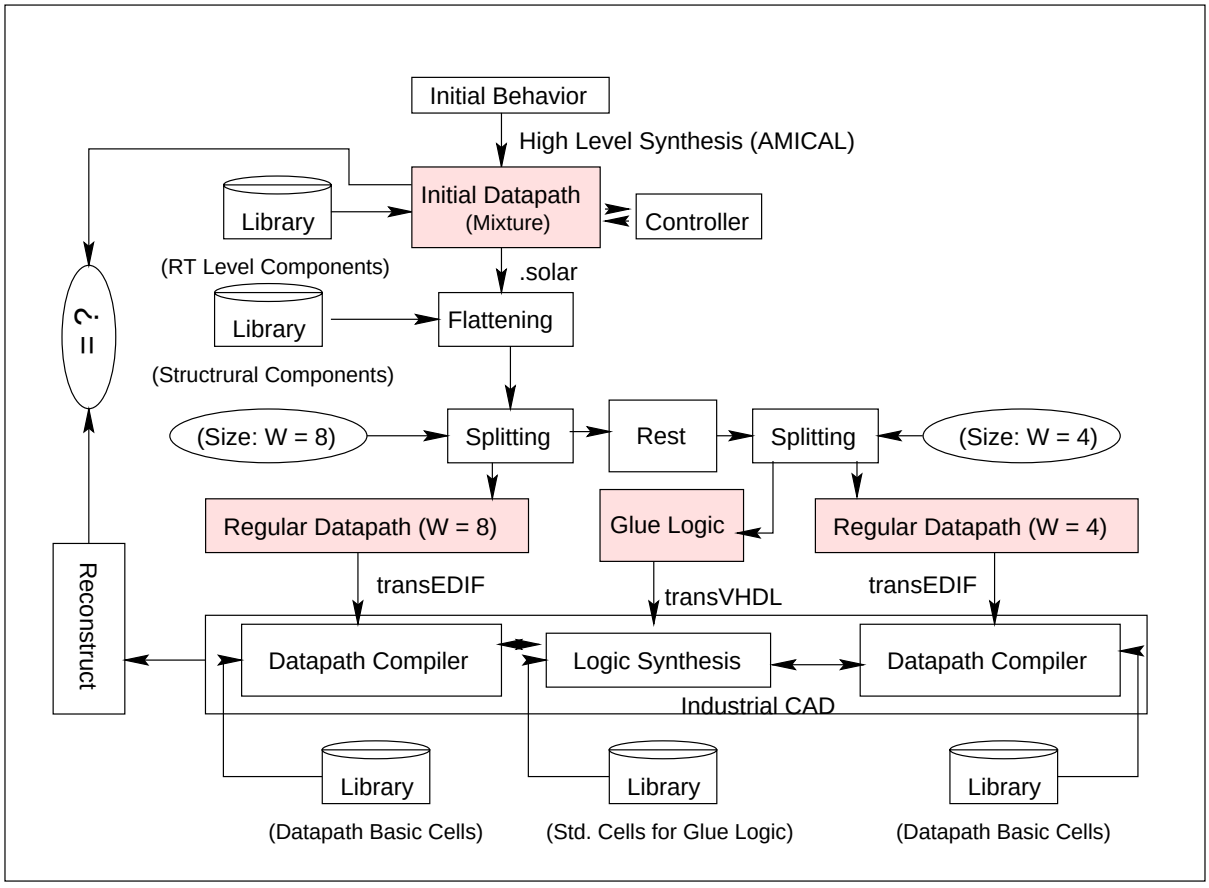


Figure 5.7: An Example undergoing a datapath decomposition

step in *Table* 5.4. The nets N_{int} and N_{inst} will be seperated after the two steps of *splitting*.

Next, we show in *Table* 5.5, the data available after completing the *splitting* step.

nb. Ports					nb. Instances			nb. Nets		
Direction		Port Type		To-tal	Library		To-tal	N_{ext}	N_{int}+ N_{inst}	To-tal
in	*out*	*control*	*data*		Datapath	Glue.Cell				
57	8	56	9	**65**	75	15	**90**	65	78	**143**

Table 5.4: List of objects of the datapath after *flattening*

We see that, the flattened datapath has been divided into 3 parts: The purpose of the two *Tables* 5.4 and 5.5 is to show that, how an initial datapath containing a larger number of ports (= 65), instances (= 90) and nets (= 143) has been broken down into three smaller blocks, each of which containing a smaller number of ports. Besides, all but the glue blocks (in this case, DP_{W8} and DP_{W4}) are parameterized and thus are kept as compact single bit schematics of datapaths.

Decomposed Objects	Ports			Instances		Nets			
	in	out	inout	Glue	Datapath	N_{feed}	N_{int}	N_{inst}	N_{ext}
DP: W=8	32	3	11	-	48	-	18	6	47
DP: W=4	18	3	6	-	27	-	24	7	27
Glue	11	2	5	15	-	-	2	0	9

Table 5.5: List of objects of the datapath after *splitting*

Next, using the *Table* 5.6, we show the details of the two regular datapaths containing the widths of eight and four bit slices, in terms of the basic functions. This facilitates a theoritical calculation of the total cell area. (recall the previous section).

Example $Area_{Controller}$ = 0.123 sq.mm (FSM synthesis) $Area_{Datapath}$ = 0.426 sq.mm (Logic synthesis)				
Library	Decomposition Results			Area
	Regular Datapaths			
Components	nb. inst/bit	Area (in terms of A_j)		in $sq.\mu$
(**Datapath**)		(W = 8)	(W = 4)	
Alu	$n_1 = 1$	$8.A_1$	$4.A_1$	13416
And-or	$n_2 = 4$	$32.A_2$	$16.A_2$	8448
Bus-stat	$n_3 = 2$	$16.A_3$	$8.A_3$	4224
Switch	$n_4 = 29$	$232.A_4$	$116.A_4$	91872
Inverter	$n_5 = 6$	$48.A_5$	$24.A_5$	6336
Ram	$n_6 = 1$	$8.A_6$	$4.A_6$	5280
Register-1	$n_7 = 1$	$8.A_7$	$4.A_7$	4224
Register-2	$n_8 = 2$	$16.A_8$	$8.A_8$	8448
	Total DP			142248
(**Glue Logic**)	nb.cells	Area using A_j		
Inverter	$n_9 = 2$	$2.A_9$	-	220
Nand gate	$n_{10} = 4$	$4.A_{10}$	-	660
	Total GLUE			880
$Area_{Datapath}$ after decomposition (cell area) = 0.143 sq.mm (after decomposition)				

Table 5.6: Results following a Datapath Decomposition

Because of the faster changing technogies, we have shown the area of the decomposed datapath in the Table-6 using symbolic values $A_1..A_10$. In addition, this helps to compute the cell area in a technology independant manner in the following way:

$$A_{Datapath} = \sum_{j=1}^{8} n_j * (W_1 + W_2) * A_j + \sum_{j=9}^{10} n_j * A_j$$

where,
$W_1 = 8$, $W_2 = 4$

$A_j (i.e., A_1 to A_8, A_9 and A_{10})$ represents either the *data* or the *glue* area occupied by a respective cell j.

We have used a 0.5μ technology in the above example. In this case, we achieved a reduction of the size of the datapath from 0.426 sq.mm to 0.143 sq.mm using the decomposition method in conjunction with an industrial datapath compiler.

5.5 Conclusion

In this chapter, we have presented a method called the *decomposition* that enables a *datapath compilation* of a flexible architecture generated by the high leevel synthesis tool Amical. The motivations were to provide an efficient datapath synthesis while allowing the initial high level specification to contain not only the standard components, but also the re-usable components of user's choice. With the advent of the high level synthesis, the datapath and its controller are automatically generated. However, most of the high level synthesis tools support a fixed functional unit scheme. On the other hand, Amical permits a flexible architecture permitting user definied re-usable components. The datapaths of such an architecture are not directly imported by a classical datapath compiler. Using the decomposition method we explain as how to use a datapath compiler, while supporting a flexible architecture at the high level. The overhead of this method is to create the structural views of each of the functional unit components, which replace the RTL views. The benefit of this method is the creation of smaller and more efficient datapaths.

We give in Appendix-7, the different C++ methods used by the Flattening and the Splitting processes of datapath decomposition together with a list of typical datapath library cell functions and their interfaces.

.

Chapter 6

TRANSLATIONS to RTL

In this chapter, we explain the translation procedures used to adapt the result of a High Level Synthesis using the AMICAL system. The goal is to convert the results available in terms of synthesized objects into equivalent data in a standard format for use by the existing CAD environment at the Register Transfer Level. The starting point is a high level synthesized description in a format called Solar *[JO92]. Solar is organized into an Object Oriented data structure in the C++ programming language. There are two translation procedures used:*

- *Solar to VHDL specifications that may be imported for the synthesis of the controller and the datapath using a RTL/logic synthesis tool. The same description can be simulated for validation, before the synthesis step.*
- *Solar to EDIF specifications suitable for datapath compilations of the regular datapaths using a classical datapath compiler.*

The term 'block box' means a component whose interface is known, but the internals hidden. Some call it black box.

.

6.1 Introduction

The generation of architecture is one of the key steps in a High Level Synthesis. The generated architecture is available at a Register Transfer Level consisting of a controller and a datapath, which is a basic model. Another architecture may be obtained, by choice, consisting of an interface between the controller and the datapath. The interface contains a set of registers to memorize the data flowing between the controller and the datapath. The architecture above is a pipelined model. The architecture must be contained in a suitable format, for the import by the lower level synthesis tools in order to complete rest of the silicon compilation. The architecture generation step must also consider the impact of the description styles on the final design. The High Level Synthesis (HLS) system AMICAL generates an architecture in a form called Solar [JO92]. In this chapter, we present the translation procedures used on the above architecture. The translated architecture is able to interract with the existing CAD environments and methodologies as well as to validate the result of a high level synthesis with respect to the initial high level specification.

6.1.1 Principle

The translation starts from the result of the high level synthesis available in the SOLAR format. The grammar of Solar used by HLS is available in [PVD95]. Solar is a compact form to store the architecture and is organized into an Object Oriented data structure in the C++ programming language. Certain translation procedures are used in order to convert each object present into an equivalent data significant in the target language. The contents of an architecture can be completely translated into VHDL due to the presence of a uniform semantics existing between:

- a solar *interface* and a VHDL *entity*.
- a solar *statetable* and a vhdl *process* controlled by *case* statements.
- a solar *contents* containing *instances* , *nets* and a VHDL architecture containing *signal declarations* and *instantiations* of *components*.

The translation to EDIF is more strait forward since Solar is EDIF-like. The contents of an architecture in the form of a regular datapath (recall Chapter-5) is structural and makes use of a target technology library. The translation of Solar *instance* becomes *cell* and *cellref* in EDIF. A *net* in solar is translated into a *net* in the EDIF.

- The EDIF *cell* contains all the ports of the Solar instance and is referred by the instance using the *cellref*.
- A solar *net* is identical with a net in the EDIF.

The components of datapath generated by HLS can be abstract in order to support the technology independance. However, mapping them onto a specific target library introduces problems such as the naming conventions, presence and/or absence of some of the ports etc. Certain symbols can be resolved by context. For example, in Solar, the *instance*s and the *nets* are distinguished by context, and thus different elements can have the same names. However, in VHDL, the *signal*s and the *instance* labels shall not have the same names, since both of them belong to the same hierarchy (architecture). This problem is overcome by adopting systematic naming convention styles.

6.1.2 Organization of Solar

Reference is made to [O'B93]. The data structure of Solar is modeled using an object oriented methodology. The Solar environment contains four components as illustrated in *Figure* 6.1.

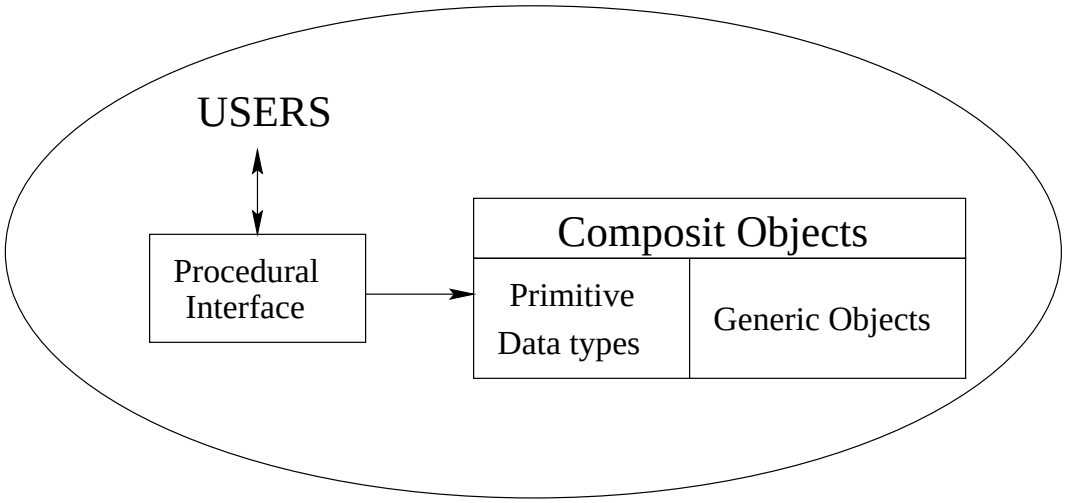
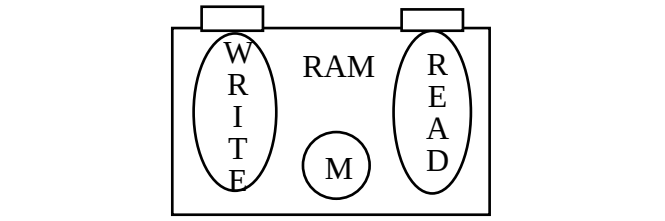


Figure 6.1: The Solar intermediate format

The first component is a set of the data type primitives. For example, the basic types for artithmetic and boolean functions. The second one contains the objects and generic lists on which all the other objects are based. The composite objects are the structures containing the primitives, some generic objects and the characteristics inherited by other existing structures. The fourth element is the procedural interface. It defines a set of functions representing unique ways to access and modify the data structure. Utilization of an Object Oriented Solar is not only easy to use, but also fecilitates future modifications.

that can easily be introduced into new systems [Par92]. An FU can be specified at different abstraction levels as shown in *Figure* 3.2. In this example, a register file called RAM is described. From a conceptual point of view, the register file is an object able to execute two operations (Read, Write) which share some data (M, *Figure* 3.2 (a)). At the behavioral level, the FU is described with a VHDL package (*Figure* 3.2 (b)) that includes two procedures and a global signal (the array RAM). Each procedure specifies the execution details of an operation. This may be a complex behavior. The FU may correspond to an existing macro-block that has already been synthesized or described at the RT level. *Figure* 3.2 (c) shows an external view of a possible realization of the register file. It is connected to two inputs (address and datain), one output (dataout) and one command (rw) that selects the procedure to execute. The high-level synthesis view of the FU summarizes the behavioral and RTL views.

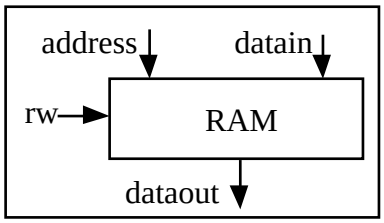


(a) Conceptual view

```
Type memory is array (0 to 3199) of integer;
Signal RAM : memory;

Procedure read (address : in integer; dataout : out integer);
Procedure write (address,datain : in integer);
```

(b) Behavioral view



(c) RTL view

```
(FU RAM
   (Area 25000) (Width 250) (Height 100)
   (Parameter (DataIn a b) (DataOut c))
   (Connector
     (DataIn address (Bit 0 12) datain (Bit 0 8))
     (DataOut dataout (Bit 0 8))
     (ControlIn rw (Bit 0 2)))
   (Operator read
     (Cycle 1
          (Active rw 1) (Transfer a address))
     (Cycle 2
          (Transfer dataout c)))
   (Operator write
     (Cycle 1
        (Active rw 2) (Transfer a address)
        (Transfer b datain))))
```

(d) High-level synthesis view

Figure 3.2: Different functional unit abstractions

It includes: the interface of the FU the FU's call parameters corresponding to the procedure parameters, the operation set executed by the FU, the micro-scheduling of each operation corresponding to data transfers at the clock cycle level. The synthesis environment can contain a library of such FUs. The designer invokes FUs through sequential procedure calls. FUs can be of any degree of complexity and can themselves be the result of a high level synthesis process.

3.2.3 Target architecture for design re-use

The target architecture of AMICAL is shown in *Figure* 3.3. It is composed of a top controller, a set of functional units and a communication network. The last two constitute the data-path. This architecture is synchronous, parallel and heterogeneous. The controller acts as a synchronous central controller and sequences the operations

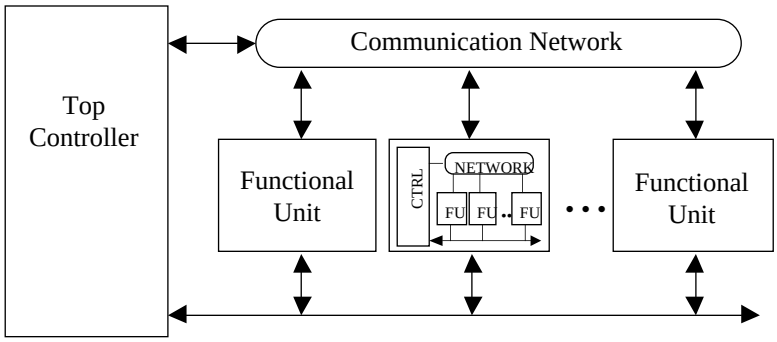


Figure 3.3: Target architecture

executed by the functional units and the communication network. The controller is generated automatically during the synthesis process. It is represented as a symbolic finite state machine. No assumption is made about its organization. At each basic cycle a new command is issued to the data-path consisting of the functional units and a communication network. A command may include the activation of several communication paths, namely, the transfers.

The architecture is parallel; it may include several functional units that may execute in parallel. The amount of parallelism is fixed during the synthesis process either automatically or manually. As explained earlier, a functional unit may execute several multi-cycle functions. The functional units interact through a communication network composed of buses, switches or multiplexers and registers. How with this scheme, large memorization blocks and I/O units are handled as functional units and managed by the user in the behavioral description will be explained further on.

The architecture is heterogeneous since it allows to mix components that may come from different design environments. A functional unit may be the result of a module generator, the result of an architectural synthesis tools such as AMICAL or CATHEDRAL, or an existing hardware resulting from a manual design. These functional units may perform different kinds of operations, such as arithmetic operations, memorization, address computation, DSP operations , I/O operation, etc. The use of this target architecture allows AMICAL to get around one of the main limitations of existing silicon compilers which often restrict the functional units of the library to simple logic and arithmetic operators. In real life circuit, it is generally the case where a design includes complex operators such as cache memories, I/O units, etc. These generally correspond to existing hardware and may execute several operations.

The datapath architecture is generated either using a bus-based or using a mux-based structure as chosen by the user. The communication network is composed of buses, switches and registers in the case of a bus-based solution, and of multiplexers and registers for multiplexer-based architectures. The network is built in order to allow the communication between functional units and with the external world. The number of buses is fixed according to parallel transfers required by the architecture (command generated by the controller). Memory with complex addressing function and specific embedded computation and control can be easily described with this scheme. The addressing functions may be realized by an independent functional unit or may be integrated within the memory unit. In the same way, complex I/O units may be used. They are also accessed through function and procedure calls, and may execute complex protocols or data conversions.

The functional unit concept of AMICAL allows the use of complex designs for re-use as functional units [KDB+95]. The present system restricts the operation executed by functional units, it requires that each operation has a staticaly predictable execution time. With such a scheme, a complex operation with a non-fixed execution time such as the *operation including data dependent computation and/or asynchronous wait statements* must be split into a set of atomic operations (e.g. start operation, check completion and get results of operation) with fixed execution time. Of course some precautions should be taken here. When the order of these operations is relevant and when the different procedures may share common data, the coherence of the results of the synthesis process may be ensured only if some directions are provided by the user. For example each set of dependent operations may be explicitly bound to the corresponding functional unit in order to have the calls to operations of a same set to be scheduled in different control steps. In the case the binding is delayed, manual re-scheduling or manual functional unit allocation may be required in order to ensure the coherence of the result.

According to the characteristics of an application, a set of functional units will be provided before starting the synthesis process. For instance, the use of special purpose hardware units able to increase the computing power of the architecture may be provided. The correspondence between the operations of the behavioral description (standard operators such as + and -, and procedure and function calls) and the functional units is made during the synthesis process. Initially we may have a library where each operation may be executed on several functional units and where each functional unit may execute several operations. We may even have operations that have different execution schemes on different functional units. The number of functional units selected (allocated or instanciated) will depend on the parallelism allowed by the initial description and the architectural transformations performed by the user during the synthesis process. The functional unit allocation performed during the synthesis process is based on an EMUCS-like algorithm [OPJC93]. It tries to share as much as possible the use of functional units. One can note that the cor-

respondence between operations and functional units can be done manually by the user.

The selection of the functional units (number and type) is a crucial step in the synthesis process. It will fix the tradeoff between the cost and the performance of the architecture. The allocation of the communication network also plays an important role in fixing this tradeoff. These steps constitute the main added value provided by high-level synthesis in contrast to RTL and logic synthesis.

3.2.4 Integrating AMICAL within existing design methodologies

AMICAL is compatible with existing logic and RTL synthesis tools. With AMICAL, it becomes possible to synthesize more complex blocks that cannot be handled efficiently by existing RTL synthesis tools. The use of a standard description language (VHDL) ensures that its inputs and outputs are also compatible with existing simulation environments. The interaction between AMICAL and existing CAD en-

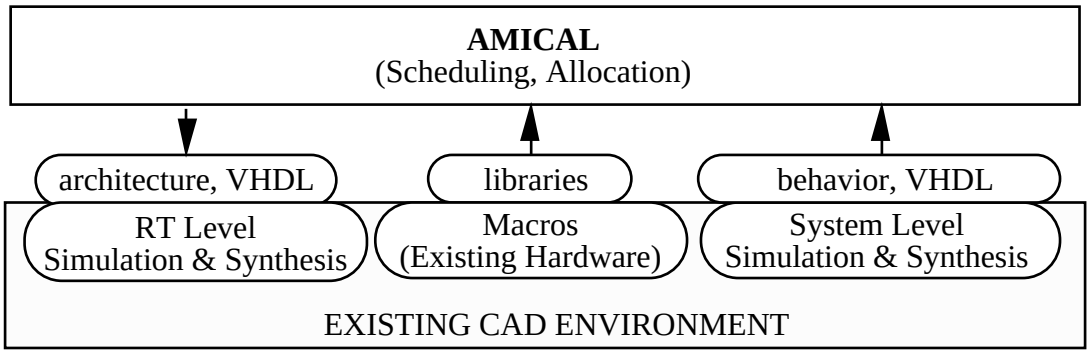


Figure 3.4: Integrating AMICAL within existing CAD environments

vironments is shown in *Figure* 3.4. Starting from a behavioral description, AMICAL performs the normal high-level synthesis steps (scheduling, allocation, etc.) and generates an RTL structure composed of a data-path and a controller.

With this scheme there are many ways in which AMICAL can be used during the design of a complex, real-time system. For example, such a design can be realized by the four following steps:

1- System level specification and verification,

2- Architectural exploration and behavioral partitioning,

3- Micro-architecture generation

4- RTL synthesis (RTL and layout synthesis environments) .

Steps 2 and 3 are performed by AMICAL. The first and the fourth steps are performed by the existing CAD environment chosen.

In the first step, the design is specified at the system level [GVNG94]. This specification may be part of a complex system or be a complete design itself. The specification should be verified through system-level simulation. The results of the simulation will be used during subsequent steps for architecture exploration. At this stage of the process, the user must select the set of FUs that will be used during architectural synthesis.

The second step includes scheduling and allocation. Once the behavioral description and the FU library are ready, AMICAL can be used for architectural exploration and synthesis. The behavioral description is partitioned into a data-path and a controller. The synthesis can be carried out automatically, manually or by a combination of both. The automatic synthesis produces the fastest architecture according to the input description. This may be modified in order to reduce the number of allocated FUs and buses.

The third step produces an RTL specification. It proceeds in producing an abstract architecture first followed by the *personalization* of this output for a given RTL synthesis and logic environment. The personalization process is programmable. It provides facilities such as handling different clocking schemes and different VHDL styles. It is detailed in Chapter-4.

Once the new VHDL specification of the architecture is generated, a validation step is needed. This step mainly involves simulating the generated description. This "post-architecture synthesis" simulation may be used to carry out a detailed performance analysis of the architecture (for example: the number of clock cycles needed to perform a given computation). Such a simulation may also be used to check the correctness of the synthesis process. This step also includes RTL and logic synthesis.

AMICAL is a pure VHDL compiler: the VHDL interpretation does not deviate from the VHDL LRM definition [VHD87]. The main advantage of this strategy is that the design may be simulated before an architectural synthesis, at the behavioral level, and after architectural synthesis, at the RT level, using the same VHDL simulation environment. During all synthesis steps, AMICAL maintains the coherence of the design at the functional level. Each manual intervention is checked first by the system and is executed only if it does not lead to violation of any kind.

3.3 AMICAL: Interactive High-Level Synthesis

This section gives a global view of the AMICAL system. More details about the algorithms used by AMICAL can be found in [JPO93, POJ92, Par92].

3.3.1 Interactive environment

AMICAL is organized as an architectural synthesis environment. It works as a design assistant, combining automatic, manual and interactive synthesis.

The interaction with the system is performed through a mouse and a graphical interface. The AMICAL user-interface consists of 3 windows:

- Control window (Top): This is generally used to show and edit the controller.

- Data-path window (Middle) : It is used to show and edit the data-path.

- Information window (Bottom): It is used to print information and error messages. It also provides information about the progress of the synthesis process such as the last command, synthesis step, contents of the other windows, etc. These information are needed in order to help the user during long synthesis sessions. This window also shows the current synthesis mode, whether Automatic or Interactive or Manual.

As we will see afterwards, one of the main strengths of AMICAL is its ability to maintain the coherence between the information included in these three windows, i.e. the different aspects of the design.

Figure 3.5 shows a screen dump that gives a flavor of AMICAL at work. The right-hand window shows a VHDL description of the algorithm (MPEG-AUDIO decoding system) being synthesized [KCBJ93]. The top window contains the transition table generated by the scheduler. The middle window shows the data-path, as synthesized by AMICAL. The bottom window provides information on the current status of AMICAL. We make use of concepts similar to those used in CORAL II [RB88]. in order to link the behavior and structure. In this way, AMICAL is able to maintain the coherence between the information contained in the three main windows. For example, the designer may require information concerning the correspondence between the controller and the data-path. In *Figure* 3.5, we have asked for information about the resources used for the execution of the two parallel operations in transition 6 of the controller representation (highlighted in top window). AMICAL has highlighted the appropriate data-path components in the middle window. The resources highlighted by AMICAL not only include the registers and the functional units necessary for storing the variables and executing the operations, but also the buses and switches used for data transfers to and from the functional units. More information is given about the control step in the bottom window. The interaction with the user is similar to the MIES system [NSM89].

3.3.2 The design-flow

The AMICAL design-flow is illustrated by *Figure* 3.6. The behavioral description is a VHDL process that may make use of complex sub-systems through procedure and function calls. However for each procedure or function used, the library used

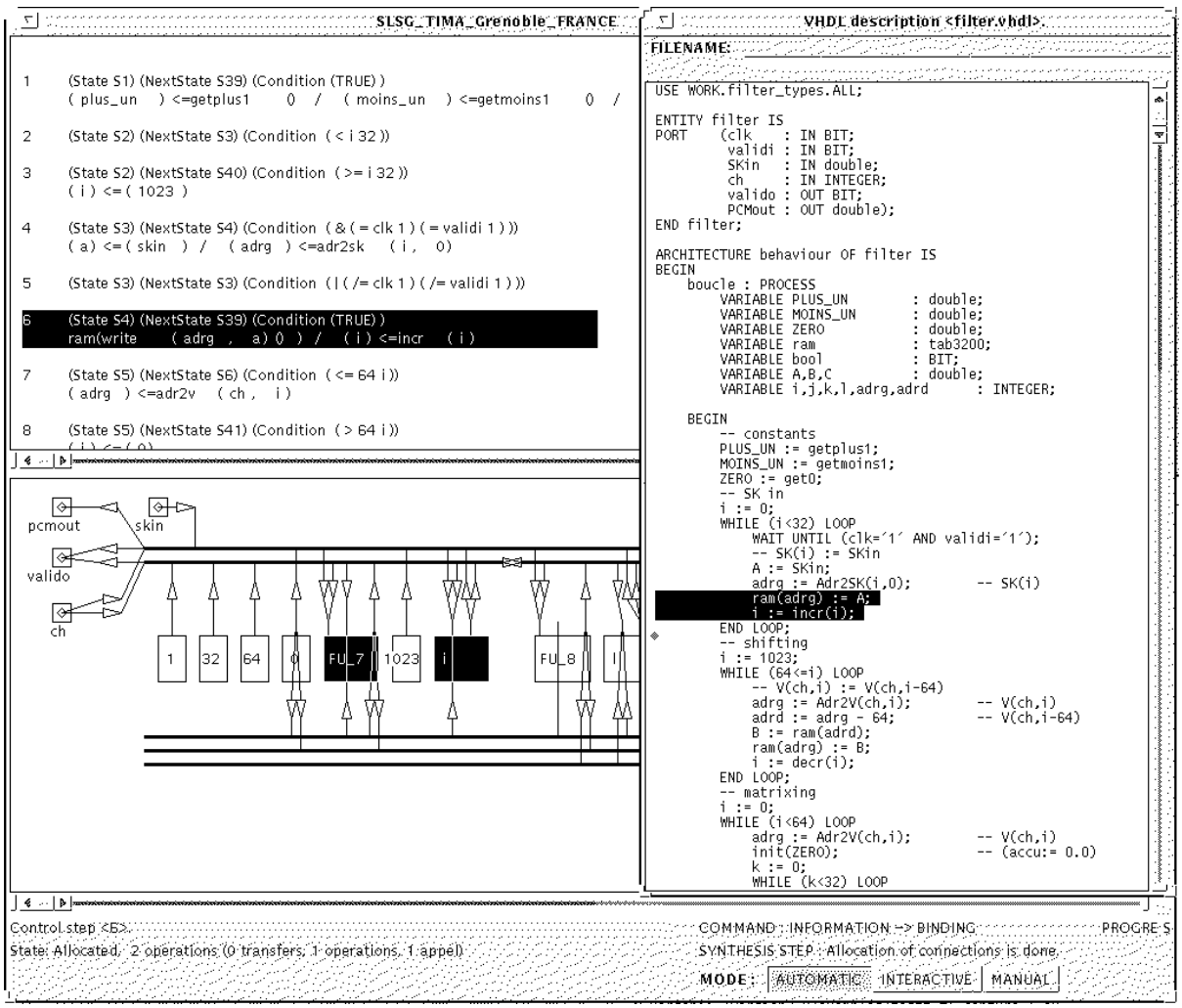


Figure 3.5: Screen Dump of AMICAL at work

as input for the behavioral synthesis must include at least one functional unit able to execute the corresponding operation. During the different steps involved in the high-level synthesis, the functional units are used as black boxes. The only pieces of information required about each functional unit are the list of procedures executed by the functional unit and some information about protocols. However to complete the description at the RT level, the details of the functional units are required. As shown in section 3.2 and by *Figure* 3.2, each functional unit can be viewed at different abstraction levels. The different steps involved in the synthesis process are: macro-scheduling, allocation, micro-scheduling and architecture generation. The macro-scheduling step makes use of the dynamic loop algorithm [ORJ93]. This is a path-based scheduling. The scheduler reads in the VHDL description and produces a finite state machine presented as a transition table. Each transition corresponds to the execution of a control step under a given condition. All the operations of a given transition may be executed in parallel. An operation may correspond to a standard operation of VHDL such as +, -, etc. or to a procedure call.

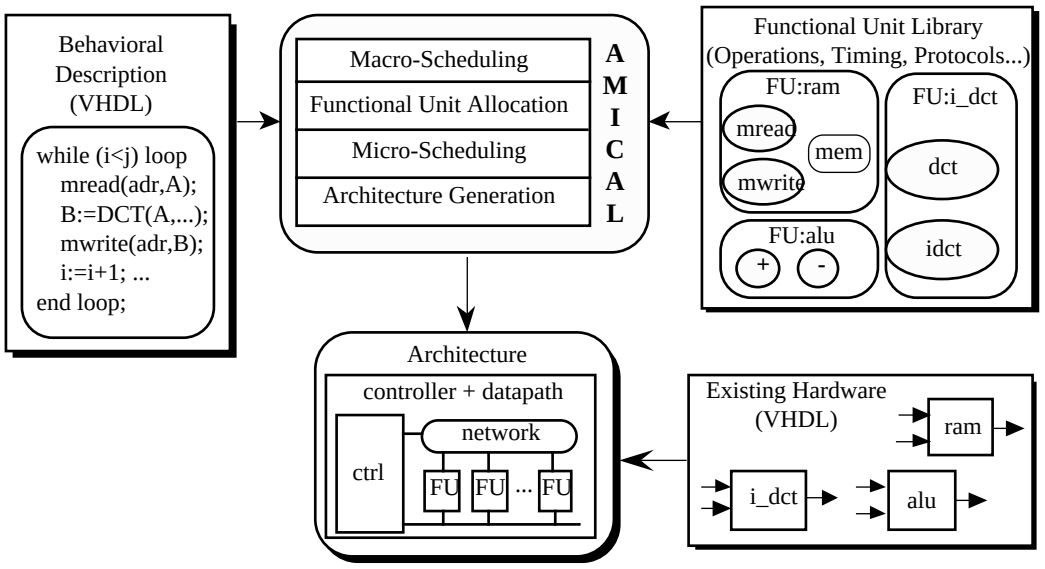


Figure 3.6: AMICAL design-flow

At this stage, each operation may take several clock cycles to execute. A transition is also called macro-cycle or macro-step. In fact a transition corresponds to a simple data-flow graph that has to be synthesized using the classical steps, which are scheduling and allocation. The goal of these steps is to refine each transition , which is also called a macro-cycle, into a set of basic control steps executing in one clock cycle each. These basic control steps are also called micro-cycles.

After scheduling, architectural synthesis starts with two kinds of information, namely the scheduled description in the form of a set of data-flow graphs and an external functional unit library. During the following steps, both allocation and binding are performed. This step makes use of an EMUCS-like algorithm [OPJC93]. The functional unit allocation step associates a functional unit with each operation in the state table. A second scheduling step ,called the micro-scheduling, is then performed according to the execution scheme for each operation; the synthesis view of the functional unit specification is given in *Figure* 4. This step makes use of an ASAP (As Soon As Possible) scheduling algorithm. Each operation is decomposed into a set of transfers, which are scheduled into micro-cycles. Each micro-cycle contains a set of parallel transfers that take one basic clock cycle to execute.

The last synthesis step is a classic architecture generation. The clock-cycle level description is mapped onto an architecture composed of a data-path and a controller. This step includes data-path generation consisting of the communication allocation, i.e. multiplexer / bus generation and the controller generation.

3.4 Generation of RTL specifications

After a high level synthesis, AMICAL provides 2 choices in order to generate the architecture in the form of RTL specifications. The 2 models available are presented in *Figure* 3.7

The basic model consists of an *interface*, a *Controller*, a *datapath* and the *interconnection*. *commands* represent a bunch of command signals for communication from the controller to the datapath indicating the action to be performed. *status* means a bunch of status signals communicated by the datapath to the controller indicating the result of action performed.

- Interface is a set of input, output and input-output ports for the communication of the architecture to the external world.
- Controller is a finite state machine of type mealy.
- Datapath is a netlist of components consisting of the functional units, communication units, memory etc.
- Interconnection is the set of signals used for the commands, status signals and connection of the datapath components to the interface.

The model with the *interface* is an extension of the basic model containing an interface between the controller and the datapath. The interface contains a set of registers that are used to temporarily hold the command and status signals. The signals are then released in a synchronized manner so as to achieve pipelining between the controller and the datapath. This way, both controller and the datapath can function simultaneously with the data available from the interface, instead of waiting for the signals. Besides, the use of interface is also envisaged for the purpose of testability using the boundary scan technique [IEEE standard].

3.4.1 Personalization

This sub section will be detailed in Chapter-4. *Personalization* provides the link to use the result of a high level synthesis in the existing CAD environments and methodologies. The problems handled by the personalization are the following:

1 Supporting different synchronization schemes

2 Handling different libraries with different naming conventions

3 Mixing synthesizable and un-synthesizable descriptions during a high level synthesis.

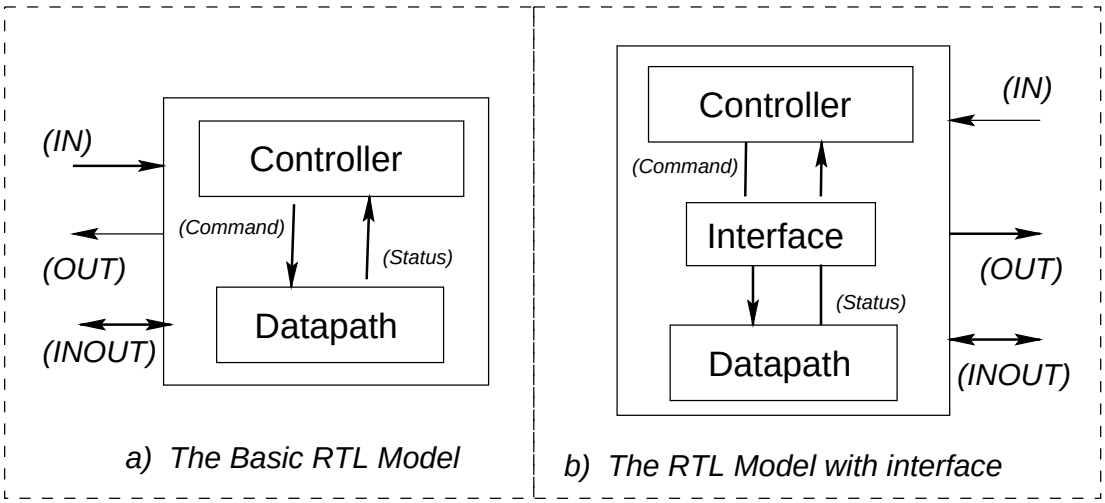


Figure 3.7: Choice of models for RTL generation

3.4.2 Datapath Decomposistion

This sub section will be detailed in Chapter-5. *Datapath Decomposition* is used to extract the *regular datapaths* for an efficient datapath synthesis using the standard datapath compilers. Without such a decomposition, the datapath is needed to be subject to a boolean optimization using logic synthesis. The problems handled by the datapath decomposition are the following:

1 Presence of complex components as compared to the basic datapath operations.

2 Presence of multi widthed data ports within the components.

3.4.3 Standard Formats

This sub section will be detailed in Chapter-6. *Standard Formats* enables the import of the architecture by the existing CAD environment and methodologies. The result of high level synthesis by AMICAL is available in an object oriented data structure called Solar [JO92]. The translators developed to interract with the existing CAD are the following:

1 Solar to VHDL translator ensuring "What You Simulate Is What You Synthesize"

2 Solar to EDIF translator ensuring the use of a given datapath compiler library.

The Controller part of the architecture is translated into an equivalent VHDL process. The Datapath is translated into an equivalent VHDL netlist for logic synthesis and also into a set of equivalent EDIF netlists for regular datapath synthesis. A

component library is used in the translations of the datapath and the interface. The component library for the datapath contains functional views for logic synthesis and the interface view for the datapath decomposition. The component library shall contain additionally, the register components (or) the scan cells for the model with interface.

3.5 Conclusions

The most important feature of AMICAL is the fact that it combines traditional behavioral-level synthesis algorithms with the ability to allow designer intervention at almost any stage of the synthesis process. The designer can simultaneously view both the control section and the data-path as well as their inter-relation. For each synthesis step, the designer can choose between automatic, manual or step-by-step execution. If either of the latter two modes is selected, AMICAL verifies that all modifications comply with a set of rules corresponding to the particular synthesis task. AMICAL also generates a statistical evaluation of each design configuration. In addition, all of the synthesis steps are performed in real-time making the system truly interactive.

The use of a standard description language namely, the VHDL ensures that the inputs and outputs of AMICAL are compatible with existing simulation environments. Additionally, it will be easy to integrate AMICAL within existing design methodologies based on logic and RTL synthesis. This integration will allow the specification and synthesis of complex systems that include behavioral blocks that may be compiled with AMICAL, and RTL blocks that may be compiled using existing logic synthesis tools. The present version produces an RTL VHDL compatible with the simulation and synthesis tools provided by Synopsys.

The link to lower level synthesis tool is ensured, thanks to an environment allowing a flexible and efficient RTL model generation. This link will be detailed in the rest of this thesis work.

Chapter 4

PERSONALIZATION

In this chapter, we introduce to the integration of an architectural synthesis tool within existing CAD environments using the concepts of architecture personalization *using a* personalization file. *The principle motivations are the following:*

1. *Migration of an unsynthesized part of a specification from high level to register transfer level,*
2. *Architecture engineering of the controller and datapath.*

The main issue of the personalization of the architecture resulting from the high–level synthesis is the generation of descriptions that are compatible with the existing RTL simulation and logic synthesis tools. Thus, the application of the method enables the mixing of already designed and existing components into a high level specification and renders the resulting architecture more flexible at the register transfer level. The personalization scheme explained in this chapter is used within the Amical *high level synthesis environment.*

.

4.1 Introduction

The *high level synthesis* (HLS) consists of two steps:

1. Transformation of a high level specification into a *controller* and a *datapath* using the steps of scheduling and allocation.

2. The generation of a *register transfer level* (RTL) description, wherein the abstract components are mapped onto components provided from the library for simulation and synthesis.

4.1.1 Background

Recently, many HLS tools are either starting to be published or introduced commercially. All those tools basically transform a high level specification expressed in the form equivalent of a VHDL [VHD87, LG88, OM96, OW96, ABOR90] process into an architecture composed of a controller and a datapath for synthesis at a Register Transfer Level, RTL. Thus, the designers are saved of their manual efforts in the above conversion. However, most of the tools had very less proliferation in to the *industry* due to their rigidity: Either the designers have no control over their designs at the RTL or the design obtained by a high level synthesis is not directly usable by the existing commercial tools used in the Industry. Another important factor is that these tools generally do not permit the mixing of already existing design parts for re-use at the high level specification. The principal objective of the method we explain in this chapter ,called, *personalization*, is to be able to support a flexible architecture.

1) To be able to mix the already existing standard and specific design components of lower level, into a high level specification such that these componenents may be migrated and synthesized at a lower level. This concept supports design re-use as well as enabling a reduced volume of initial specification.

2) Provide a method, using which, the synthesized architecture at the register transfer level may be customized. This solution supports a flexible architecture at the RT level instead of being a 'push-button' architecture. The personalization tool permits the user to specify the requirements in the form of a *personalization file* to be detailed later. A personalization file uses a simple list based format and is designed in such a way as to extend with the future requirements.

4.1.2 Motivations

A lot of research has been done concerning the synthesis at the high level, while the RTL description generation has not been considered a lot during the last years.

Nevertheless, this is a key step in making HLS widely accepted and used. Although conceptually simple, the architecture generation step introduces a few problems that have to be solved in order to render the HLS practical:

1. *Handling different libraries with different naming conventions:*
   The generated architecture generally uses several components, like registers, multiplexers and the other functional units. During the HLS we use generic components that must be mapped onto the existing components during the architecture generation step.
   The library components may have specific naming conventions. For example the name of the clock and the reset signals are not standardized and identical in all the libraries; We may have components in the libraries that use different names for these signals.

2. *Supporting different synchronization schemes:*
   As the result of a HLS consists of an architecture containing a controller and a datapath, there are several schemes for the synchronization of these two components. We must thus customize the synchronization scheme [1], for the RT level. Therefore we may use external link signals for the datapath or we may use a specific clock distribution block that has to be added to the circuit.

3. *Mixing synthesizable and Unsynthesizable descriptions:*
   The behavioral description accepted by HLS tools consists generally of a single process that belongs to a large system specification. After the HLS, the resulting architecture has to be re–connected to the rest of the system. Besides, several aspects concerning the interfaces (such as personalized port drivers or pads) are difficult to handle at the behavioral level.

The three above mentioned problems require a large flexibility from the architecture synthesis generation step. Unfortunately most of the existing HLS tools assume predefined solutions for most of these problems.
The first problem got lots of attention in the VSS system [Dut88]. The authors defined a generic library allowing an automatic generation of the architecture for a further lower level synthesis. Although this solution provides some flexibility, it doesn't allow the re-use of existing libraries.
The synchronization problem is harder to handle in a flexible way. Therefore, most of the existing HLS tools are using one or several prefixed synchronization models.
Mixing synthesizable and un–synthesizable descriptions can be solved automatically [Syn94] in most cases. However it remains a few cases where, for example, the personalization of the I/O pads, may need the addition of extra cells which are still hard to be solved in a completely automatic manner.

[1] for engineering the architecture by providing hardware related details

4.2 The Architecture Personalization

The concept of an architecture personalization is introduced by *Figure* 4.1.

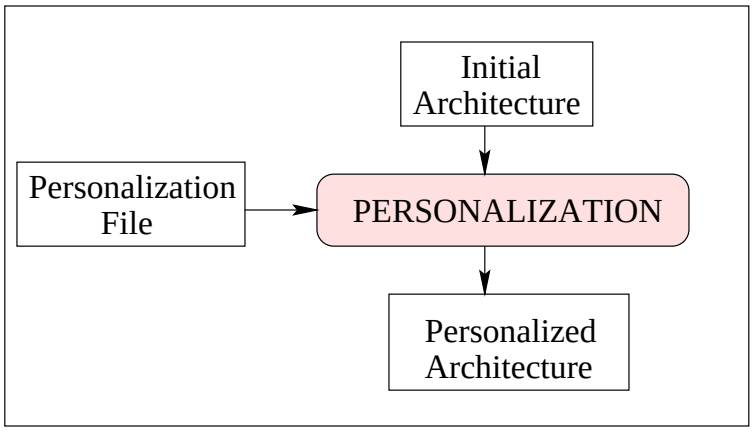


Figure 4.1: Personalization Concept

Starting from an initial architecture available from a HLS and a personalization file, this process allows to produce a personalized architecture. The personalization file is a script written in a list based format which describes the transformations required on the initial architecture. The typical transformations include:

- Addition of Extra Components
- Addition of Extra Signals
- Addition of Local Interconnects

The Personalization step makes the generation of RTL descriptions more flexible. More importantly, it provides a way to mix synthesizable and unsynthesizable descriptions in the high level specifications as well as to have several synchronization models at the RT level.

4.2.1 Architecture

An architecture consists of a set of components communicating either virtually or through a user-defined synchronization scheme. The components are either behavioral or structural. The latter can be either hierarchical or flat. A synchronaization scheme consists of signal(s) for synchronization, such as the clock(s), reset, a mechanism for synchronization. The minimum clock period takes into account the physical delay required by the operations of the components and of the communications.

The behavioral part of the architecture available after a high level synthesis is a *controller*, which in the case of AMICAL is a finite state machine of type *Mealy* and

the structural part is a *datapath* inter-connecting the functional units used within. The functional units are required for data holding(Registers), data operations(Ex: Adder), data communication (bus, mux) and so on.
In addition, registers are used in between the controller and the datapath for pipelining which may also be used for the testing such as by using the IEEE boundary scan.

4.2.2 Solar format

The results of a high level synthesis with *Amical* are represented using an EDIF-like format called *Solar*. The Solar format is internally managed in an object oriented manner containing the methods for the access and operations on the different objects. The data in the Solar format is classified by views. For example, a "communicating" view represents a black box (component interface); a "behavior" view is used to represent a FSM (controller); the view "structure" is used to represent a netlist (datapath). The communicating view has only the ports and is the basic subset of any other views. Both the views behavior and structure are divided into interface (of ports) and contents. More details on the Solar format may be found in [PVD95] and detailed information on the personalization file (also known as a *Global file*) are available from [Vij94c]. A *Personalization file* acts as a means for performing the personalization and is in a list based format. It enables an easier link between the results of *Amical* after a high level synthesis with the existing design tools at the register transfer level for simulation and synthesis.

4.2.3 Abstract Component Library

The *Abstract Library* contains the components (or Functional Units) in the Solar format. For the purpose of *personalization*, it is sufficient to have only the communicating view of each of them, that provides an interface without the implementation details. If a component is already present in an another view (either structure or behavior), personalization scheme would consider only the interface part, ignoring all the rest (i.e., the contents). Thus, the abstract library supports either hiding of the functionality if desired or postponing an implementation. An example of a component description in an abstract library is given below:

```
(solar MUX
   (designunit Mux_2
      (view contains_port_only (viewtype "communicating")
            (interface
               (port (array in1 8)  (direction in) (bit)
                                    (property resolved BusX))
               (port (array in2 8)  (direction in) (bit)
```

```
                                    (property resolved BusX))
                (port (array out1 8) (direction out) (bit)
                                    (property resolved BusX))
                (port Sel            (direction in) (bit)
                                    (property portType control)
                                    (property resolved BusX))
  ) ) ) )
```

4.3 The Personalization Method

The *Personalization* is initiated using a *personalization file* and a set of components present within a SOLAR *abstract library*. The *personalization* is applied on an abstract architecture which is available in the form of solar files separately for the controller and the datapath. The result is a personalized architecture in a solar format readily translatable into a VHDL format, thus enabling access to the synthesis and simulation tools at the register transfer level.

A functional Unit would be searched within the abstract library whenever, a new *InstanceRef* is encountered within the Solar design file [2] or whenever a *glue cell* is identified in the personalization file.

4.3.1 The Personalization file

The goal of architecture personalization is to translate a *design file* available in an intermediate (Solar) format into another Solar file incorporating the *personalization* details coming from the *Personalization File*. Here is part of a *Personalization File* that personalizes the *gcd_circuit* obtained after a HLS using Amical.

```
(global gcd.global
    (block gcd_circuit
       (signal clk (dir in) (attr clock) (type std_Ulogic))
       (signal rst (dir in) (attr reset) (type std_Ulogic))
       (signal cont_clock   (dir  internal) (attr clock)
                            (type std_Ulogic))
       (signal cont_reset   (dir  internal) (attr reset)
                            (type std_Ulogic))
       (signal data_clock   (dir  internal) (attr clock)
                            (type std_Ulogic))
       (signal data_reset   (dir  internal) (attr reset)
```

[2] *datapath* and *circuit* are demonstrative examples containing the *Instancerefs*.

```
                        (type std_Ulogic))
      (glue   S)
)   )
```

- All signals of direction *in* [3] are added, after having ensured for any duplication, as the equivalent of *ports* of a VHDL entity: and thus *clk* and *rst* in this case.
- All signals of direction *internal* are added as equivalents of *signals* in the VHDL architecture: and thus *cont_clock*, *cont_reset*, *data_clock* and *data_reset* in this case. These signals are used, in order to map the formal port names of the components.
- All definitions of type *glue* would become equivalent to a VHDL component definition and use: and thus a component named *S* would be added equivalent in terms of a (VHDL) component declaration and a component instantiation in this case. The *interface* or port(s) of the component would be fetched from the *Solar Library*. The use of glue is the way to migrate an unsynthesized component from a high level specification.

The architecture resulting from the Personalization is shown by *Figure* 4.2, in which each of the blocks, namely the parts control, data, circuit, the glue cell (S) are independant VHDL entities communicating together.

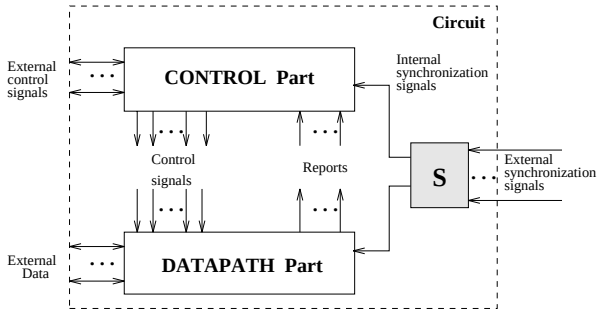


Figure 4.2: An architecture after a personalization

4.4 The Personalization Actions

The results of an architectural synthesis session with AMICAL are files in the Solar format describing an abstract architecture. These files can be personalized by the designer using the information contained in the Personalization file in order to

[3] similarly, the other possible directions, namely: *out, inout, buffer* if present.

.