High-Level Synthesis of Control and Memory Intensive Applications

Peeter Ellervee

Stockholm 2000

Thesis submitted to the Royal Institute of Technology in partial fulfillment of the requirements for the degree of Doctor of Technology

Ellervee, Peeter
High-Level Synthesis of Control and Memory Intensive Applications

ISRN KTH/ESD/AVH--2000/1--SE
ISSN 1104-8697
TRITA-ESD-2000-01

Royal Institute of Technology
Department of Electronics
Electronic System Design Laboratory
Electrum 229, Isafjordsgatan 22-26
S-164 40 Kista, Sweden
URL: `http://www.ele.kth.se/ESD`

Abstract

Recent developments in microelectronic technology and CAD technology allow production of larger and larger integrated circuits in shorter and shorter design times. At the same time, the abstraction level of specifications is getting higher both to cover the increased complexity of systems more efficiently and to make the design process less error prone. Although High-Level Synthesis (HLS) has been successfully used in many cases, it is still not as indispensable today as layout or logic synthesis. Various application specific synthesis strategies have been developed to cope with the problems of general HLS strategies.

In this thesis solutions for the following sub-tasks of HLS, targeting control and memory intensive applications (CMIST) are presented.

An internal representation is an important element of design methodologies and synthesis tools. IRSYD (Internal Representation for System Description) has been developed to meet various requirements for internal representations. It is specifically targeted towards representation of heterogeneous systems with multiple front-end languages and to facilitate the integration of several design and validation tools.

The memory access bandwidth is one of the main design bottlenecks in control and data-transfer dominated applications such as protocol processing and multimedia. These applications are often characterized by the use of dynamically allocated data structures. A methodology has been developed to minimize not only the total number of data-transfers by merging memory accesses, but also to minimize the total storage by reducing bit-width wastage.

An efficient control-flow based scheduling strategy, segment-based scheduling, has been developed to avoid the typical disadvantages of most of the control-flow based schedulers. The scheduler avoids construction of all control paths by dividing control graph into segments during the graph traversal. Segmentation drastically reduces the number of possible paths thus speeding up the whole scheduling process.

Fast and simple heuristic algorithms have been developed which solve the allocation and binding tasks of functional units and storage elements in a unified manner. The algorithms solve a graph coloring problem by working on a weighted conflict graph.

A prototype tool set has been developed to test HL S methodology of CMIST style applications. Some industrially relevant design examples have been synthesized by using the tool set. The effectiveness of the methodologies was validated by comparing the results against synthesis results of commercially available HLS tools and in some cases against manually optimized designs.

To my parents.

Acknowledgements

I would like to begin by expressing my gratitude to Professor Hannu Tenhunen for providing me with the opportunity to do research in a very interesting area. I would also like to send huge thanks to my supervisor, Docent Ahmed Hemani, for providing me with the challenging tasks and problems to solve, and for the help and support.

Of the other people who work here at Electronic System Design Laboratory I would like to thank Dr. Axel Jantsch for his support and for is invaluable comments about various research topics. Dr. Johnny Öberg for keeping up a constant competition between us, for all the help and, especially, for all the fuzzy discussions.

Prof. Anshul Kumar and Prof. Shashi Kumar of Indian Institute of Technology in New Delhi for their extremely valuable collaboration, for the suggestions and discussions, which significantly helped to improve various parts of this work. Prof. Francky Catthoor and Miguel Miranda from IMEC, Leuven, for their help and cooperation in the memory synthesis area.

Dr. Mehran Mokhtari from the high-speed group for all the interesting discussions and for convincing me “that transistor can work”. Prof. Andres Keevallik and Prof. Raimund Ubar from Tallinn Technical University for introducing to me the sea of research in the area of digital hardware. Dr. Kalle Tammemäe from TTU for the help and cooperation.

Special thanks to Hans Berggren, Julio Mercado and Richard Andersson at the system group for keeping all the computers up and running, to Aila Pohja for keeping an eye on things when she was here, and to Lena Beronius for keeping an eye on things around here now. I would also like to thank all colleagues at ESDlab, at the high-speed group, at TTU, and all my friends for their support and friendship.

Last but not least I would like to thank my parents for their care, and for the inspiring environment they offered to me and to my brother.

Kista, February 2000
Peeter Ellervee

Table of Contents

List of Publications

High-Level Synthesis

1. A. Hemani, B. Svantesson, P. Ellervee, A. Postula, J. Öberg, A. Jantsch, H. Tenhunen, “High-Level Synthesis of Control and Memory Intensive Communications System”. Eighth Annual IEEE International ASIC Conference and Exhibit (ASIC’95), pp.185-191, Austin, USA, Sept. 1995.

2. B. Svantesson, A. Hemani, P. Ellervee, A. Postula, J. Öberg, A. Jantsch, H. Tenhunen, “Modelling and Synthesis of Operational and Management System (OAM) of ATM Switch Fabrics”. The 13th NORCHIP Conference, pp.115-122, Copenhagen, Denmark, Nov. 1995.

3. B. Svantesson, P. Ellervee, A. Postula, J. Öberg, A. Hemani, “A Novel Allocation Strategy for Control and Memory Intensive Telecommunication Circuits”. The 9th International Conference on VLSI Design, pp.23-28, Bangalore, India, Jan. 1996.

4. J. Öberg, J. Isoaho, P. Ellervee, A. Jantsch, A. Hemani, “A Rule-Based Allocator for Improving Allocation of Filter Structures in HLS”. The 9th International Conference on VLSI Design, pp.133-139, Bangalore, India, Jan. 1996.

5. P. Ellervee, A. Hemani, A. Kumar, B. Svantesson, J. Öberg, H. Tenhunen, “Controller Synthesis in Control and Memory Centric High Level Synthesis System”. The 5th Biennial Baltic Electronic Conference, pp.397-400, Tallinn, Estonia, Oct. 1996.

6. P. Ellervee, A. Kumar, B. Svantesson, A. Hemani, “Internal Representation and Behavioural Synthesis of Control Dominated Applications”. The 14th NORCHIP Conference, pp.142-149, Helsinki, Finland, Nov. 1996.

7. P. Ellervee, A. Kumar, B. Svantesson, A. Hemani, “Segment-Based Scheduling of Control Dominated Applications in High Level Synthesis”. International Workshop on Logic and Architecture Synthesis, pp.337-344, Grenoble, France, Dec. 1996.

8. J. Öberg, P. Ellervee, A. Kumar, A. Hemani, “Comparing Conventional HLS with Grammar-Based Hardware Synthesis: A Case Study”. The 15th NORCHIP Conference, pp.52-59, Nov. 1997, Tallinn, Estonia.

9. P. Ellervee, S. Kumar, A. Hemani, “Comparison of Four Heuristic Algorithms for Unified

Allocation and Binding in High-Level Synthesis". The 15th NORCHIP Conference, pp.60-66, Tallinn, Estonia, Nov. 1997.

10. P. Ellervee, S. Kumar, A. Jantsch, A. Hemani, B. Svantesson, J. Öberg, I. Sander, "IRSYD - An Internal representation for System Description (Version 0.1)". TRITA-ESD-1997-10, Royal Institute of Technology, Stockholm, Sweden.

11. A. Jantsch, S. Kumar, I. Sander, B. Svantesson, J. Öberg, A. Hemani, P. Ellervee, M. O'Nils, "Comparison of Six Languages for System Level Descriptions of Telecom Systems". First International Forum on Design Languages (FDL'98), vol.2, pp.139-148, Lausanne, Switzerland, Sept. 1998.

12. P. Ellervee, S. Kumar, A. Jantsch, B. Svantesson, T. Meincke, A. Hemani, "IRSYD: An Internal Representation for Heterogeneous Embedded Systems". The 16th NORCHIP Conference, pp.214-221, Lund, Sweden, Nov. 1998.

13. P. Ellervee, M. Miranda, F. Catthoor, A. Hemani, "Exploiting Data Transfer Locality in Memory Mapping". Proceedings of the 25th Euromicro Conference, pp.14-21, Milan, Italy, Sept. 1999.

14. P. Ellervee, M. Miranda, F. Catthoor, A. Hemani, "High-level Memory Mapping Exploration for Dynamically Allocated Data Structures". Subm. to the 36th Design Automation Conference (DAC'2000), Los Angeles, CA, USA, June 2000.

15. P. Ellervee, A. Kumar, A. Hemani, "Segment-Based Scheduling of Control Dominated Applications". Subm. to ACM Transactions on Design Automation of Electronic Systems.

16. P. Ellervee, M. Miranda, F. Catthoor, A. Hemani, "Optimizing Memory Access Bandwidth in High-Level Memory Mapping of Dynamically Allocated Data Structures". Subm. to IEEE Transactions on Very Large Scale Integration (VLSI) Systems.

HW/SW Codesign and Estimation

17. A. Jantsch, J. Öberg, P. Ellervee, A. Hemani, H. Tenhunen, "A software oriented approach to hardware-software co-design", (poster paper). International Conf. on Compiler Construction (CC-94), pp.93-102, Edinburgh, Scotland, April 1994.

18. A. Jantsch, P. Ellervee, J. Öberg, A. Hemani, H. Tenhunen, "Hardware-Software Partitioning and Minimizing Memory Interface Traffic". In Proc. of the European Design Automation Conference (Euro-DAC'94), pp.226-231, Grenoble, France, Sept. 1994.

19. P. Ellervee, A. Jantsch, J. Öberg, A. Hemani, H. Tenhunen, "Exploring ASIC Design Space At System Level with a Neural Network Estimator". Seventh Annual IEEE International

ASIC Conference and Exhibit, (ASIC'94), pp.67-70, Rochester, USA, Sept. 1994.

20. P. Ellervee, J. Öberg, A. Jantsch, A. Hemani, "Neural Network Based Estimator to Explore the Design Space at System Level". The 4th Biennial Baltic Electronic Conference, pp.391-396, Tallinn, Estonia, Oct. 1994.

Other papers

21. M. Mokhtari, P. Ellervee, G. Schuppener, T. Juhola, H. Tenhunen, A. Djupsjöbacka, "Gb/s Encoder/Decoder Circuits for Fiber Optical Links in Si-Bipolar Technology". International Symposium on Circuits and Systems (ISCAS'98), pp.345-348, Monterey, USA, May 1998.

22. A. Djupsjöbacka, P. Ellervee, M. Mokhtari, "Coder/Decoder Using Block Codes". Pending patent, Ericsson E08867.

23. J. Öberg, P. Ellervee "Revolver: A High-Performance MIMD Architecture for Collision Free Computing". Proceedings of the 24th Euromicro Conference, pp.301-308, Västerås, Sweden, Aug. 1998.

24. J. Öberg, P. Ellervee, A. Hemani, "Grammar-based Modelling of Clock Protocols for Low Power Implementations: A Case Study". The 16th NORCHIP Conference, pp.144-153, Lund, Sweden, Nov. 1998.

25. T. Meincke, A. Hemani, S. Kumar, P. Ellervee, J. Öberg, T. Olsson, P. Nilsson, D. Lindqvist, H. Tenhunen, "Globally Asynchronous Locally Synchronous VLSI Architecture for large high-performance ASICs". International Symposium on Circuits and Systems (ISCAS'99), Orlando, Florida, USA, May 1999.

26. A. Hemani, T. Meincke, S. Kumar, A. Postula, T. Olsson, P. Nilsson, J. Öberg, P. Ellervee, D. Lundqvist, "Lowering Power Consumption in Clock by Using Globally Asynchronous Locally Synchronous Design Style". The 36th Design Automation Conference (DAC'99), pp.873-878, New Orleans, LA, USA, June 1999.

27. E. Dubrova, P. Ellervee, "A Fast Algorithm for Three-Level Logic Optimization". International Workshop on Logic Synthesis, Lake Tahoe, CA, USA, June 1999.

List of Abbreviations

AFAP	As Fast As Possible
ASIC	Application Specific Integrated Circuit
ASM	Algorithmic State Machine
ATM	Asynchronous Transfer Mode
BFSM	Behavioral Finite State Machine
BG	Basic Group
BNF	Backus-Naur Form
CAD	Computer Aided Design
CDFG	Control Data Flow Graph
CFG	Control Flow Graph
CMIST	Control and Memory Intensive SysTems
CPU	Central Processing Unit
DADS	Dynamically Allocated Data Structure
DAG	Directed Acyclic Graph
DFG	Data Flow Graph
DLS	Dynamic Loop Scheduling
DP	Data Path
DSP	Digital Signal Processing
DTI	Data Transfer Intensive
ESG	Extended Syntax Graph
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FU	Functional Unit
GSA	Graph Scheme of Algorithm
GSM	Global System for Mobile Telecommunications
HDL	Hardware Description Language
HLS	High Level Synthesis
ILP	Integer Linear Programming
IR	Internal Representation
IRSYD	Internal Representation for System Description
LDS	Loop Directed Scheduling
NP	Non-Polynomial
OAM	Operation And Maintenance
PBS	Path Based Scheduling
PMM	Physical Memory Management
PSM	Program State Machine
RT	Register Transfer

RTL	Register Transfer Level
SDL	System Design Language
SOC	System-On-a-Chip
SPP	Segment Protocol Processor
VHDL	VHSIC Hardware Description Language
VLSI	Very Large Scale Integration
XFC	eXtended Flow Chart

1. Introduction

The evolution of chip technology is following guidelines of the famous Moore's Law [Moo96], according to which the most important characteristics are improving twice in every 18 months. At present time, the process people are continuing to achieve these parameters and the production lines have been able to fulfil the predictions. A problem is that the development of design tools can not keep up with this chip development, though there have tremendous improvements in VLSI CAD tools.

1.1. High-level synthesis

Recent developments in microelectronic technology, namely sub-micron technology, allow implementing a whole digital system as a single integrated circuit. The introduction of System-on-a-Chip (SOC) requires new design methodologies to increase designers productivity significantly. Another important factor is time-to-market which has been decreasing over the last years. That means that there exists a demand for efficient design methodologies at system-level and high-level.

CAD technology has been developing also rapidly during the last decades. This has been caused both by the growing size of task, i.e. chips to be produced, and by the availability of increasingly powerful computers to solve these tasks. The advantages in general, and especially the use of higher level of abstractions for specification, are being pushed by the following factors:

- The need to decrease the time to design a VLSI circuit and to get it to the market. This time is getting shorter and shorter due to reduce of lifetime of the product.
- The increased complexity of circuits, amplified by the improvements in the processing technology.
- The increased cost of design iteration - from specification to fabrication - requires less error prone design methods.
- The availability of fast prototyping approaches have somewhat lessened effects of the increased iteration cost, but they add more iterations in effect.
- Increased use of special circuit technology types, e.g. ASICs, FPGAs, require also special methodologies for efficient implementation.
- It is often necessary to retarget an existing design to a new technology. The design process can be simplified also by reusing parts (components) of existing older designs.
- Related to the reuse is the cost of maintenance, i.e. it is essential that the specification is easy to read and is well documented.

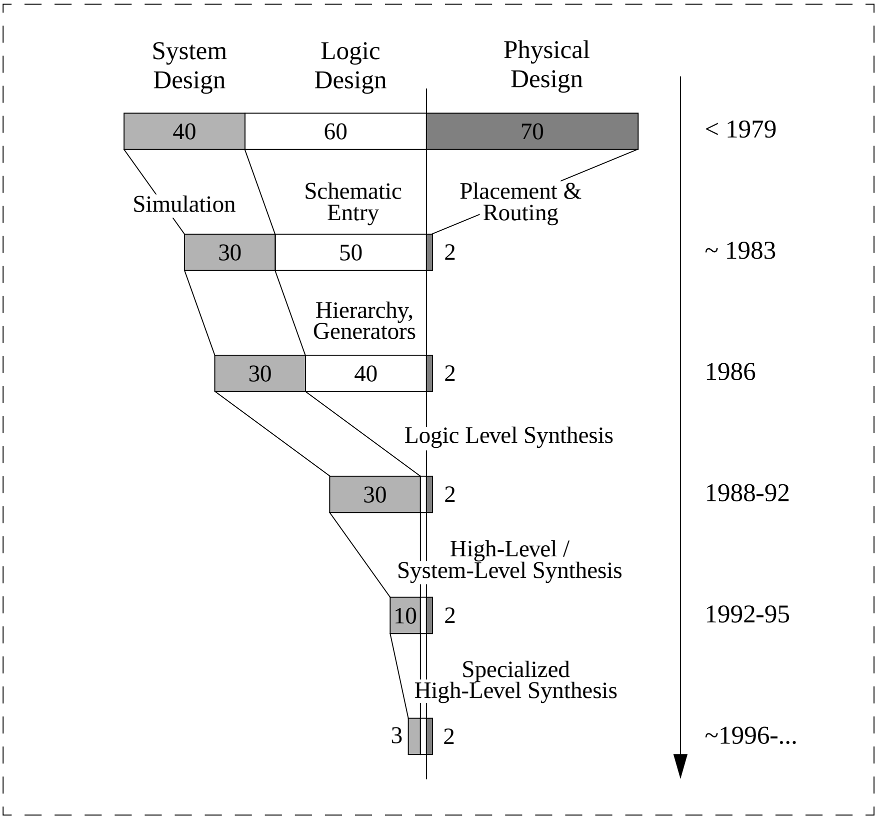


Figure 1.1. Efforts needed to design a 20 kgate circuit (Modified from [MLD92])

Before 1979, about 40% of design efforts was spent in the physical design phase when designing a 20 kgate circuit [MLD92]. In four years, placement and routing tools arrived onto the market and thus effectively reducing the physical design phase from 70 to 2 person-months. The other improvements plus simulators reduced the system and logic design phases by 20% in total. The introduction of hierarchy and hardware generators by 1986 had reduced the logic synthesis phase another 10 person-months. The emergence of logic synthesis tools, between 1988 and 1992, cut this figure down to two. Around 1995, with introducing of high-level synthesis (HLS) the system level synthesis phase had been reduced to 10 person-month. In recent years, various specialized HLS tools have allowed further reduce of synthesis times. The design phases and their reduction over time are illustrated in Figure 1.1.

The phrase *high-level* implies that the functionality of the circuit to be synthesized, or a part of it, is specified in terms of its behavior as an algorithm. Due to this, HLS is also called as *behavioral-synthesis*. The relation between HLS and the other synthesis phases is best described by

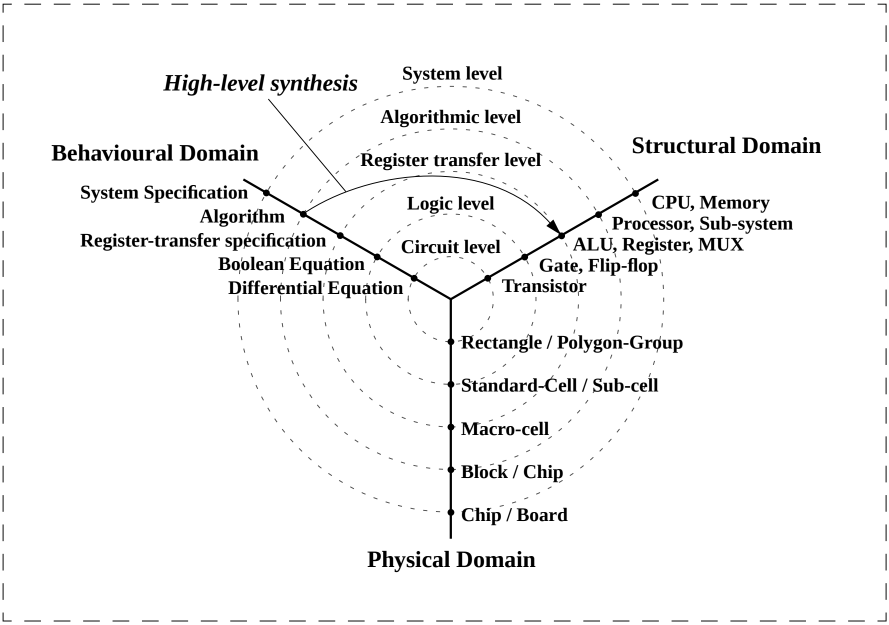


Figure 1.2. Description domains and abstraction levels

using the well-known Y-chart, introduce by Gajski and Kuhn [GaKu83]. The three domains - behavioral, structural and physical - in which a digital system can be specified, are represented as three axes (see Figure 1.2). Concentric circles represent levels of abstraction. A synthesis task can be viewed as a transformation from one axis to another and/or as a transformation from one abstraction level to another. HLS is the transformation from the algorithmic level in the behavioral domain to the register transfer level in the structural domain. The following enumerates how CAD tools benefit from HLS ([Hem92], [EKP98]):

- Automatization simplifies handling of larger design and speeds up exploration of different architectural solutions.
- The use of synthesis techniques promises correctness-by-construction. This both eliminates human errors and shortens the design time.
- The use of higher abstraction level, i.e. the algorithmic level, helps the designer to cope with the complexity.
- An algorithm does not specify the structure to implement it, thus allowing the HLS tools to explore the design space.
- The lack of low level implementation details allows easier re-targeting and reuse of existing specifications.
- Specifications at higher level of abstraction are easier to understand thus simplifying maintenance.

HLS is usually divided into four different optimization tasks, namely partitioning, allocation, scheduling and binding. Some authors skip the partitioning task, seeing it as a task on a higher abstraction level, e.g. system synthesis; or skip the binding task, seeing it as a sub-task of the allocation. A brief description of these four tasks follows ([GDW93], [DMi94], [EKP98]):

- *Partitioning* divides a behavioral description into sub-descriptions in order to reduce the size of the problem or to satisfy some external constraints.
- *Allocation* is the task of assigning operations onto available functional unit types available in the target technology.
- *Scheduling* is the problem of assigning operations to control steps in order to minimize the amount of used hardware. If performed before allocation (and binding), it imposes additional constraints how the operations can be allocated.
- *Binding* assigns operations to specific instances of unit types in order to minimize the interconnection cost.

All of the above mentioned tasks are very hard to solve exactly because of their combinatorial nature. For practical purposes, it is sufficient to have good enough solutions in reasonable time. Various heuristic and evolutionary algorithms have been proposed to solve these hard optimization tasks, e.g. [GDW93], [DMi94], [Lin97], [EKP98].

1.2. HLS of control dominated applications

Although HLS has been successfully used in many cases, it is still not as indispensable today as layout or logic synthesis. Despite the last decade of research, there is still a long way to go before the HLS tools can compete with and exceed human designers in the quality of the produced hardware. The designers, at the same time, will be able to work at higher abstraction levels. HLS research has been focused in the past on partitioning, scheduling and allocation algorithms. Some real designs, including some DSP architectures, are very simple and HLS has been rather successful, especially in synthesis of data-dominated applications. However, HLS does not consist only of scheduling and allocation. It involves converting a system specification or a description in terms of computations and communications into a set of available system components and synthesizing these components. The main problems can be outlined as follows, in principle ([GDW93], [HSE95], [Lin97], [BRN97]):

- there exist a need for application specific synthesis strategies which more efficiently could cooperate with the features of a specific application domain;
- existing internal representations can not encapsulate details of a specification without some loss of information, therefore more general representations are needed;
- the need for efficient estimation algorithms, especially in the deep sub-micron area where wiring dominates both gate delay and chip area.

HLS experiments with designs which are dominated not by the data flow but control flow and

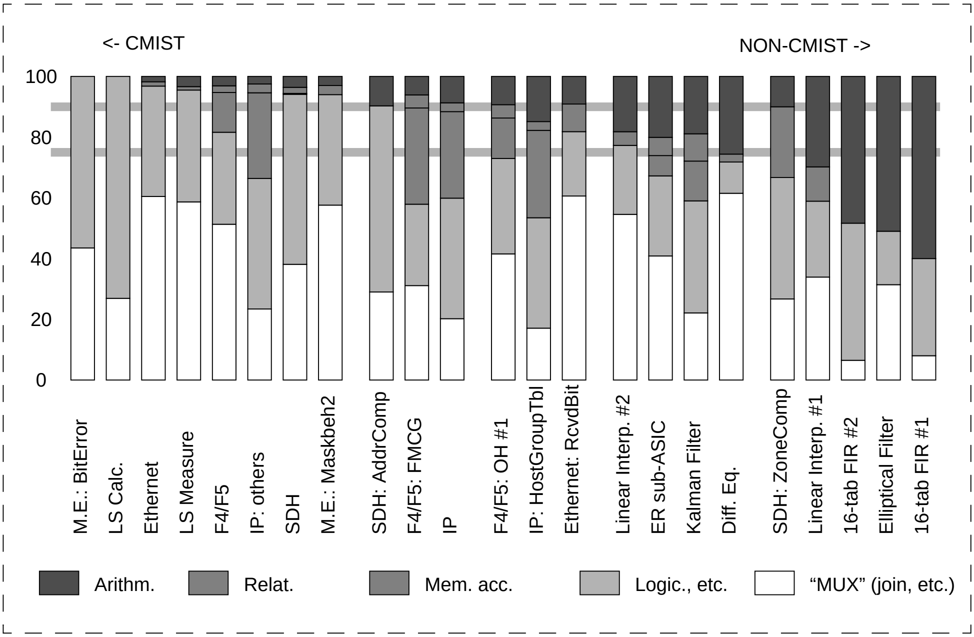


Figure 1.3. Distribution of operation types

data transfers have pointed out that the traditional, data flow oriented synthesis strategy does not work well ([HSE95], [SEP96]). A different approach is needed which would take into account the main characteristics of Control and Memory Intensive Systems (CMIST). The principal strategy for area optimization of data flow oriented approaches has been the reuse of RTL components, especially arithmetic functional units. However in CMIST the number of arithmetic operations is small compared to the number of control, relational and memory access operations. Figure 1.3. illustrates the relative amount of different operations in various design examples. The operations have been divided into five distinct groups - arithmetic, relational, memory access, logic and data transfer (“MUX”) operations. The design examples are sorted in such a way that their “CMIST-ness” is increasing from right to left, i.e. the arithmetic dominated designs are in the right side.

This has resulted in a synthesis methodology where only operations, which are candidates for HLS reuse strategy, are kept in the data path. Other operations such as relational operations with constant operands or memory indexing operations are moved to controller or allocated to the specialized address generator. The efficacy of the methodology was shown by applying it to an industrial design and comparing results from two commercial HLS tools. [HSE95], [SEP96]

Another successful specialization in HLS is targeting interface and protocol synthesis. Although different approaches are used, all methodologies use some kind of formal description to specify the design (see, e.g. [PRS98], [Obe99])

Development of domain specific HLS tools has, though, a drawback. The more domain-specific a tool is, the smaller its customer base will be. A possible solution is to embed domain specific knowledge from wider areas. [Lin97]

1.3. HLS sub-tasks targeted in the thesis

The initial work with the CMIST applications showed that there exist many domain specific problems, which need to be solved. Solutions for some of the problems are proposed in the thesis. The specialized methods were developed to cover the following sub-tasks:

- internal representation which encapsulates primarily the control flow to support CMIST synthesis strategy;
- pre-packing of data fields of dynamically allocated data structures to optimize the memory bandwidth in data transfer intensive (DTI) applications;
- segment-based scheduling of control dominated applications which targets as-fast-as-possible (AFAP) schedule and takes a special care of loops;
- fast heuristics for unified allocation and binding of functional units and storage elements.

A prototype tool set was designed to validate the developed methods. The tool set, the overall design flow used by the tool set and the target architecture are described in chapter 2.

With the help of the tool set some industrial design examples were synthesized. The results were compared with the synthesis results of commercial HLS tools and with the manually optimized design, if available. The results of sub-tasks are discussed in the related chapters. The synthesis results of industrially relevant design examples are presented in chapter 7. Some points of interest are discussed in details in the same chapter.

Each of the sub-tasks is briefly introduced in the following sub-sections.

1.3.1. Internal representation for system description (IRSYD)

Every research laboratory has developed some specific methodology to deal with system-level and high-level synthesis problems. An intermediate design representation or an Internal Representation (IR) is always an important element in these methodologies, many times designed just keeping in mind the convenience of translation and requirements of one tool (generally synthesizer or simulator). An IR is also important to represent specifications written in various high level languages like VHDL, C/C++, etc. Development of complex digital systems requires not only integration of various types of specifications, but also integration of different tools for analysis and synthesis. Effectiveness of the integration relies again on the IR.

Design of an appropriate IR is therefore of crucial importance to the effectiveness of a CAD environment. To preserve the semantics of the original specification in any language, and at different synthesis stages, an IR must have features to describe the following:

- static and dynamic structural and functional modularity and hierarchy;
- sequence, concurrency, synchronization and communication among various sub-systems;
- representation of abstract data types;
- mechanism for representing tool dependent information;
- mechanism for communication among various tools; and
- reuse of design or behaviors.

IRSYD, described in chapter 3., was developed to meet these requirements. Unlike many other IRs, IRSYD is specifically targeted towards representation of heterogeneous systems with multiple front-end languages and to facilitate the integration of several design and validation tools. IRSYD is used as the IR of the prototype HLS tool set, presented in the thesis. It forms the heart of the Hardware-Software codesign environment under development at ESDlab ([ONi96]), which will allow specification in a mix of several languages like SDL, Matlab, C/C++, etc.

The syntax of IRSYD was developed keeping in mind requirements for efficient parsing. The lack of reserved words significantly simplifies mapping front-end languages into IRSYD. This makes it very uncomfortable for designers to read and/or modify it, of course, but the effectiveness of loading and storing is much more important for any IR. The principles and structure of IRSYD are reported also in [EKJ97], [EKJ98].

1.3.2. Pre-packing of data fields in memory synthesis

Memory has always been a dominant factor in DSP ASICs and researchers in this community where one of the firsts to address the problems related to memory as an implementation component. Storage requirements of control ASICs are often fulfilled by registers and rarely require large on chip memories on the same scale as the DSP ASICs. Control applications that require large storage were in the past implemented in software. These days, more and more of control applications are implemented in hardware because of the performance requirements.

The complexity of protocol processing applications, its history of being implemented in software and demands on productivity motivate use of dynamically allocated data structures (DADS) while specifying and modeling such applications. Matisse [SYM98] is a design environment under development at IMEC that addresses implementation of such structures and other issues related to the implementation of protocol processing functionality.

DADS are defined on the basis of functional and logical coherence for readability. Retaining such coherence while organizing the DADS physically in the memory does not optimize the required storage nor does it optimize the required bandwidth. These optimization goals are addressed by analysis of the dependencies between accesses and using them as the basis for packing elements of DADS into a single memory word. Incorporated into Matisse’s physical memory management phase the pre-packing step has the following benefits:

- minimizes the total number of data-transfers;
- minimizes the total storage by reducing bit-width wastage;
- reduces search space complexity for the rest of the physical memory management flow;
- reduces addressing complexity.

The methodology, presented in chapter 4., combines horizontal and vertical mapping of data elements of arrays onto physical memories. It relies mainly on the data available at compile time, although profiling information can be used for fine-tuning. The pre-packing methodology is also reported in [EMC99], [EMC00].

1.3.3. Segment-based scheduling

Scheduling is one of the key steps in High Level Synthesis (HLS) and a significant portion of the HLS research has been devoted to the problem of scheduling. The scheduling algorithms can be classified into two major categories: data-flow based and control-flow based. Data-flow based scheduling algorithms allow to use flexible cost functions and exploit parallelism in data dependencies. What they lack is the chaining of operations and treatment of control operations. The control-flow based schedulers allow, in principle, operator chaining and target mostly as-fast-as-possible schedules. Typical disadvantages of control-flow based schedulers is their complexity because they try to optimize all possible paths and they handle loops as ordinary paths. [Lin97]

It has been realized that efficient control-flow based scheduling techniques must take into account the following issues:

- *Control constructs*: The control dominated circuits typically have a complex control flow structure, involving nesting of loops and conditionals. Therefore, in order to get a high quality design, the entire behavior inclusive of the control constructs needs to be considered, rather than just the straight-line pieces of code corresponding to the basic blocks. In particular, loops deserve a special attention in order to optimize performance.
- *Operation chaining*: In control dominated circuits, there are large numbers of condition checking, logical and bit manipulation operations. The propagation delays associated with these operations are much smaller as compared to arithmetic operations. Therefore, there is a large opportunity for scheduling multiple operations (data as well as control) chained together in a control step and ignoring this would lead to poor schedules.

Several schedulers have been developed for HLS of control dominated systems but there is no scheduler, which solves the problem comprehensively and efficiently, considering all the relevant issues. The segment-based scheduler, described in chapter 5., meets essentially these requirements. The scheduling approach avoids construction of all control paths (or their tree representation). This is achieved by dividing control graph into segments during the graph traversal. Segmentation drastically reduces the number of possible paths thus speeding up the whole scheduling process because one segment is analyzed at a time only.

The initial version of the segment-based scheduling was reported in [EKS96b].

1.3.4. Unified allocation and binding

Allocation and binding of functional units, storage elements and interconnections are important steps of any High Level Synthesis (HLS) tool. Most of the tools implement these steps separately. Generally, the binding task can be mapped onto one of the two graph problems, namely, clique partitioning or graph coloring. If the task can be represented as a conflict graph then the binding can be looked as a graph coloring problem. If the task is represented as a compatibility graph, then the binding can be looked as a clique partitioning problem. Unfortunately, optimal solutions to these graph problems require exponential time and, therefore, are impractical. A large number of constructive algorithms have been proposed to find fast but non-optimal solutions to these problems. [GDW93], [DMi94]

In most of the HLS approaches and systems, storage binding, functional unit binding and interconnection binding problems are done separately and sequentially. It has been long realized that a unified binding, though harder, can lead to a better solution. Heuristic algorithms, presented in chapter 6., for solving the allocation and binding problems in a unified manner were developed for the prototype HLS tool set. The algorithms solve graph coloring problem by working on a weighted conflict graph. The unification of binding sub-tasks is taken care by the weights. The weights and cost functions model actual hardware implementations therefore allowing reliable estimations. The algorithms differ in the heuristic strategies to order the nodes to be colored.

In CMIST applications, there are very few arithmetic operations and, therefore, there is very little scope of reuse or sharing of functional units. This simple fact validates the use of simple heuristics since a good enough solution can be found very fast when there exist very few possible bindings. The initial versions of the algorithms ware reported in [EKH97].

1.4. Conclusion and outline of the thesis

Advantages in the development of microelectronic technologies allow creating of chips with millions and millions of transistors. The need for fast and efficient design methodologies has been the pushing force in the development CAD tools. This is especially true in the recent years when time-to-market has been equally important than area or speed of a design. The tendency is that it is getting more and more important with every day.

High-level synthesis has been one of the ways to speed up the design process. Unfortunately, a general HLS approach is still a dream and several domain specific approaches have been developed. In this thesis, solutions for some of the sub-tasks of HLS for control and memory intensive applications are presented. The sub-tasks cover the following topics:

- control flow oriented internal representation;
- methodology for memory mapping optimization by exploiting locality of data transfers;
- efficient control-flow based scheduling algorithm;
- fast heuristics for unified allocation and binding.

The thesis is organized as follows. In chapter 2., the prototype HLS tool set designed to tests the solutions listed above is presented. In chapter 3., the internal representation (IRSYD) used also by the prototype tool is presented. In chapter 4., the methodology for pre-packing of data fields of dynamically allocated data structures is described. The segment-based scheduling algorithm is presented in chapter 5. In chapter 6., fast heuristics for unified allocation and binding of functional units and storage elements are described. HLS results of industrially relevant design examples are discussed in chapter 7. Conclusion are summarized, and future research directions are discussed in chapter 8.

2. Prototype High-Level Synthesis Tool “xTractor”

This chapter gives an overview of “xTractor” - a prototype HLS tool set dedicated for CMIST style applications. The overall synthesis flow and target architecture are defined. The whole structure of the tool set and component tools are described.

2.1. Introduction

The tool set, described in this chapter, was developed to test High-Level Synthesis methodology of CMIST style applications. The nature of applications, i.e. control and data transfer dominance, defined the overall synthesis flow and transformations needed to convert a behavioral description of a design into RTL description synthesizable by commercial tools. The tool set consists of several smaller component programs to solve certain synthesis steps, and of an interactive shell around the component programs. The name “xTractor” is a mixture from “extractor”, i.e. FSM extractor, and “tractor”, i.e. a robust but powerful tool in its field. FSM extractor itself points that the main task in the synthesis CMIST style applications the main task is to generate an efficient FSM ([HSE95], [SEP96]).

All component programs input CDFG in synthesizable subset of IRSYD ([EKJ97], [EKJ98], described in chapter 3.), manipulate it and output the resulting CDFG. The subset allows only a single clocked process, which is sufficient to test the synthesis methodologies. Handling of more than one process can be done with extra tools, which compose and decompose IRSYD modules. The modular structure of the shell makes it easy to add such additional tools into the overall synthesis flow. The synthesizable subset of IRSYD corresponds semantically to its predecessor XFC ([EHK96], [EKS96]) and some of the component tools still work internally with XFC. All component tools were designed initially to work with XFC and they are gradually converted to use IRSYD.

Two of the component tools are used as input and output of the tool set. One of the programs generates subset of IRSYD from Rat-C [BeMe84], a subset of C. The availability of the source code of the compiler and its very simple structure allowed with very little effort to map the compiler’s output onto IRSYD. An earlier VHDL to IRSYD translator relied on SYNT-X ([Hem92], [HSE95]) which is not available anymore. SYNT-X was a research derivative of the SYNT HLS system marketed by Synthesia AB, Stockholm, Sweden. New translators, from VHDL to IRSYD and from SDL to IRSYD are under development in Department of Computer Engineering at Tallinn Technical University, Estonia, and in Electronic System Design Laboratory at Royal Institute of Technology, Sweden, repetitively.

The second component tool generates RT-Level VHDL ([KuAb95]) or Verilog ([SSM93]) for Logic Level synthesis tools. Synthesis scripts in dc-shell can be generated optionally for Synopsys Design Compiler ([Syn92b]). Different styles, selectable by options, target better exploitation of back-end synthesis tools.

The component tools, which work only with IRSYD, are as follows:

- *Data-path handler* analyzes performance related properties of the CDFG and performs some simpler data-path optimizing transformations. The analysis covers delay, area and performance estimations. The data-path optimizations simplify CDFG with the goal to able more efficient execution of other tools.
- *Memory extractor* lists and/or maps arrays onto memories. The tool generates also so-called memory mapping files where the mapping of arrays onto memories can be modified (fine tuned by the designer). This tool helps to evaluate memory synthesis related issues described in chapter 4. Full implementation of the methodology requires more efficient DP analysis and optimization, and incorporation of some physical memory management steps described in section 4.3.
- *State marker* generates states while traversing the control flow of the IRSYD. This component tool implements the segment-based scheduling approach presented in chapter 5. Various constraints - clock period, segment's length, marking strategies, etc. - can be applied to guide the state marking. Another version of the state marking tool (*SYNT-X state marker*) implements the state marking strategy of the original CMIST approach [HSE95], [SEP96]. This tool is faster than the main state marker but may generate significantly inferior results.
- *Allocator/binder* allocates and binds operations and variables into functional units and registers. Interconnections (multiplexers) are allocated and bound together with related functional units and/or registers. This tool is an implementation of fast unified allocation and binding heuristics described in chapter 6.
- *Check CDFG* is a supporting tool. It allows to check the integrity of CDFGs and reports some relevant parameters.

The interactive shell organizes the overall synthesis flow; i.e. in which order and with which parameters the component tools are called. There exist four predefined synthesis flow styles with different degrees of designer activity. Additional styles can be created and stored as projects.

The component tools are written in C/C++ ([SaBr95]) and can be ported to several platforms. The interactive shell is written in Tcl/Tk ([Ous94], [Rai98], [TclTk-www]), a scripting language available for multiple platforms. There are approximately 30,000 lines of C/C++ code and 3,300 lines of Tcl/Tk scripts. The whole tool set has been compiled and tested on HP-UX A.09, SunOS 4.1 and 5.5, and Linux 2.2.

The overall synthesis flow and transformations applied onto data-path are described in section 2.2. The target architecture is discussed in section 2.3. The structure of the tool set is described in sections 2.4. and 2.5.

2.2. Synthesis flow

The overall synthesis flow, used in xTractor, is similar to the synthesis flow of any HLS approach ([GDW93], [DMi94], [EKP98]). The three main steps can be outlined as follows:

- in the *partitioning* phase memories are extracted from the initial behavioral description as separate components;
- operations are assigned to states (control steps) during *scheduling* phase; and
- unified *allocation and binding* assigns operations to specific functional units.

Some simpler data-path optimizations (transformations), described below, can be applied before (or after) every main step. Most of the synthesis steps can be skipped. The scheduling phase is an exception because it is the only step that inserts state marks into the behavioral control flow. The whole synthesis flow is illustrated in Figure 2.1. The simplest flow consists of three steps - IRSYD generation, state marking and RT level HDL generation. All other steps can be applied iteratively in any order. Memories can be extracted separately, one-by-one, each of the extractions followed by constant propagation, for instance. This allows to create multiple intermediate solutions for design space exploration. The scheduling step allows to insert extra state marks into already scheduled CDFG by rerunning with tighter constraints. This allows combining of manual and automatic state markings.

Although there exist a large number of data-path transformations, only the most obvious ones have been implemented. The main problem is that most of the transformations improve a part of the design while introducing a penalty in another part. Good examples are speculative execution of operations and elimination of common sub-expressions (e.g. [RaBr96], [LRJ98], [EKP98], [WDC98]). In both cases, a result can is calculated and stored for later usage if it is beneficial, i.e. the cost of temporary storage is cheaper than (re)calculation, for instance. The need for cost evaluation requires also good estimation methods. These and many other transformations can be made at the source code level of the design to be synthesized and are therefore not implemented in the prototype tool set. The transformations are planned to be implemented in future together with related estimation techniques.

The two well known compiler oriented transformations ([ASU86], [GDW93], [EKP98]), listed below, have been implemented:

- *Constant propagation* - operations with constants are replaced with the result of the operation. This transformation is applied for all data types and operations used in the synthesizable subset of IRSYD - boolean, signed and unsigned operands, and arithmetic, logic and relational operations.
- *Variable propagation* tries to reduce the total number of variables by eliminating copies of variables.

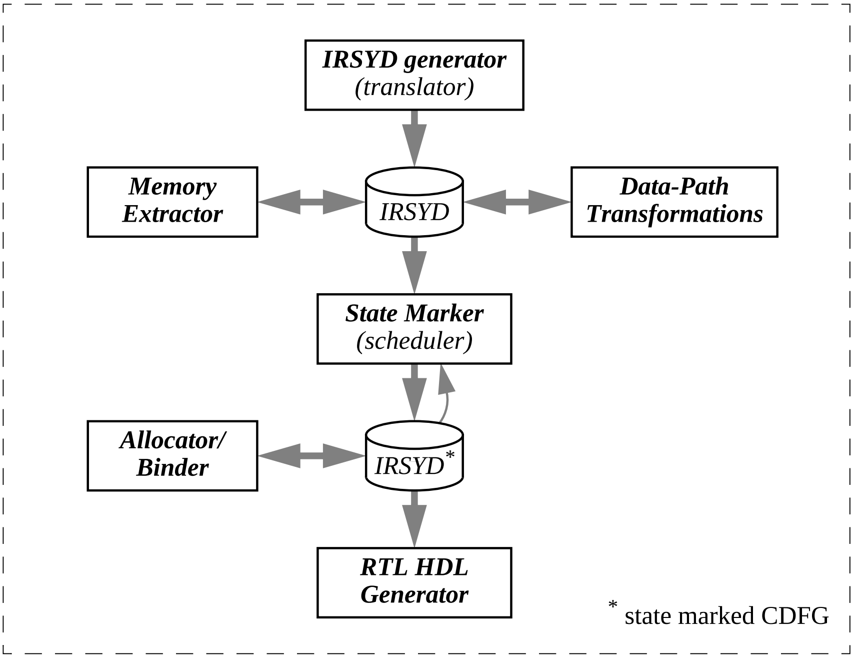


Figure 2.1. Synthesis flow

A logic operation may be replaced with a simpler one during the constant propagation - a NAND operation with constant ‘1’ is replaced with NOT operation, etc.

The third transformation, *simplification of operations* with constants, is actually a combination of compiler oriented, flow-graph and hardware specific transformations. It is applied to multiplications or divisions with constant. An operation is replaced with a combination of shift- and add-operations when the cost estimations show improvement. A multiplication, for instance, is replaced when the constant has three or less active bits. This, of course, assumes good estimates of the hardware. The problem is that at this phase it is almost impossible to estimate reuse of functional units.

All these transformations were easy to implement but their existence allows to write more readable and therefore less error prone behavioral codes.

Experiments with designs have shown that there exist some transformations more, which would be worth of incorporating into xTractor. Converting control flow operations into data flow operations, e.g. replacing conditional branches with a set of logic operations, would simplify state marking step. Complex nested if-statements, even when scheduled into a single control step, may create a local path explosion thus unnecessarily increasing the scheduling time. It may be beneficial also to replace a complex set of logic operations with a combination of if-statements and simpler sets of operations to allow scheduling of these operations over multiple clock steps. [BRN97], [EKP98]

Automatized partitioning of the CDFG into a set of sequentially working sub-CDFGs is the second useful transformation. The main goal of the partitioning would be to simplify later optimization steps.

These transformation can be applied onto the source code, of course, and this have been the main reason why the transformations have not been implemented yet.

The correctness of the synthesized design is based on assumptions that the synthesis transformations are correct. The transformations were validated using simulations on behavioral and RT levels.

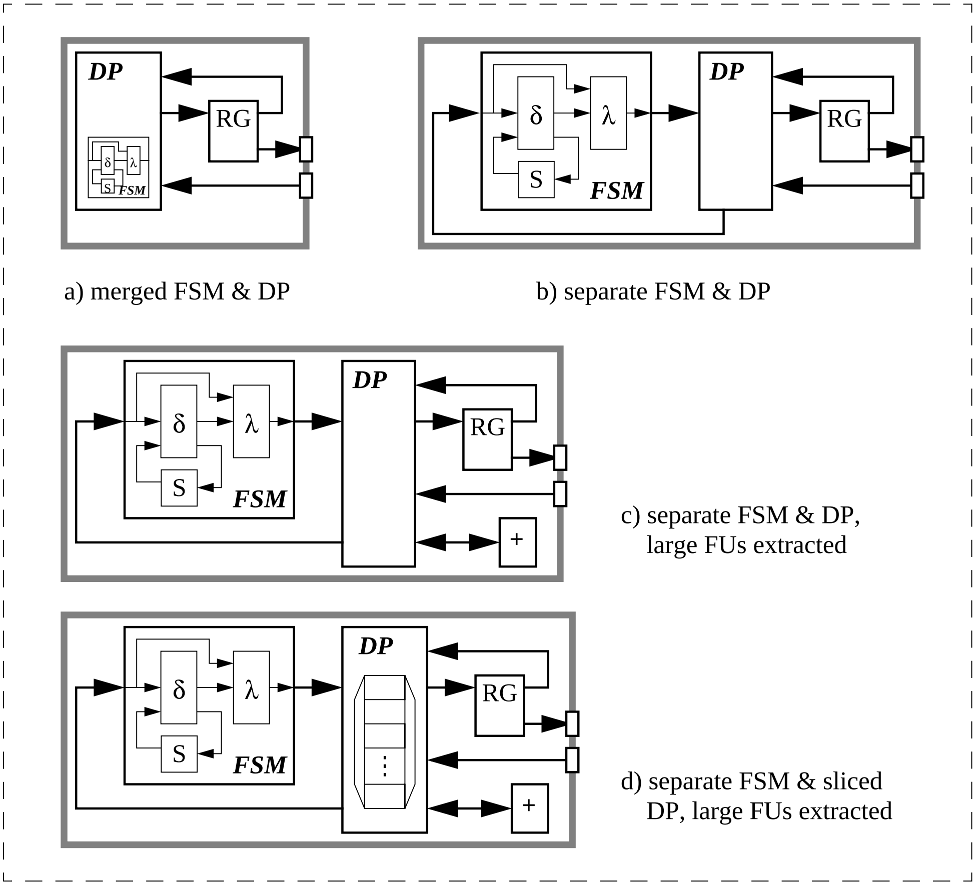
2.3. Target architecture

The last step of the prototype tool is the outputting of RT level HDL (VHDL or Verilog) code of the design. The target architecture is defined by advantages of logic synthesis tools and characteristics of the CMIST applications. [Syn92b], [HSE95]

- Modern logic level synthesis tools can handle rather complex descriptions consisting of state machines, storage elements, structured and random logic. Such a style where data-path operations are mixed with control operations but a distinct FSM still exists, is sometimes referred to as Behavioral RTL. For fast and efficient synthesis, though, these different styles should be still segregated, i.e. there should be sharp boundary in the HDL description between random logic and storage elements, etc. Global optimization techniques are very often prohibitively expensive and the use of local optimizations, e.g. optimizing module by module, gives comparable results in significantly faster synthesis times.
- CMIST applications have very few large operations worth of reusing and they can be identified easily. The rest of the design consists of operations - logic, relational, etc. - which can be very efficiently optimized by logic optimizations techniques. It can be said that from the traditional HLS point of view the operations have been moved into controller.

Keeping arithmetic operations (functional units) free of implementation details, i.e. as abstract as possible, allows better exploitation of the back-end tools - a specific architecture is selected depending on the synthesis constraints.

Four different architectures can be generated by xTractor. The actual style is selected by designer when executing the HDL generating component tool. Unfortunately there are no clear rules which style to select in which case since each of the styles allows better exploitation of some of the optimizations but may unnecessarily complicate other optimization tasks. The four architec-



a) merged FSM & DP

b) separate FSM & DP

c) separate FSM & DP, large FUs extracted

d) separate FSM & sliced DP, large FUs extracted

Figure 2.2. Target architecture

tures are listed below (see also Figure 2.2).

- Merged FSM and DP (Figure 2.2.a). Data-path is essentially a part of the next-state and output functions of the state machine. Efficient logic and FSM optimizations are applied to the whole design. This architecture should be used only for units with narrow data-path and without arithmetic operations to avoid state explosion.
- Separate FSM and DP (Figure 2.2.b). An explicit FSM has been separated from the rest of the design and the data-path is presented as a single combinatorial logic block. This allows to optimize FSM and DP separately. This architecture is the best suited for wider data-paths without arithmetic operations.
- Separate FSM and DP, large FUs extracted (Figure 2.2.c). Larger functional units, mostly arithmetic units, are extracted from the data-path. The extracted FUs, detected by the allocation and binding tool, are typically also reused. The main benefit of this architecture is that regular logic, i.e. the arithmetic units, has been separated form the random logic, i.e. the rest

of the DP.

- Separate FSM and sliced DP, large FUs extracted (Figure 2.2.d). Experiments with different design examples showed that the back-end logic synthesis could be speedup significantly when splitting the data-path into smaller parts (slices). Such a partitioning increases locality of optimizations without worsening the result quality - operations from very different parts of an algorithm are seldom reused and therefore can be optimized separately.

It is possible also to generated structures where the FSM and DP are kept together but functional units have been extracted and the combined data-path has been partitioned into slices. These structures should be avoided since they do not allow to use FSM optimization techniques because of the size of the equivalent state machine.

The main idea behind the DP slicing is to force the logic synthesis tools to narrow the search space for reuse. Although the tools usually have an option, which allows to trigger the degree of such a reuse the actual effect is rather small. Table 2.1. presents synthesis results of three different designs. All designs were synthesized in four different ways:

- full data-path with default allocation (reuse on);
- full data-path without default allocation (reuse off);
- sliced data-path with default allocation; and
- sliced data-path without default allocation.

Table 2.1. Synthesis times for different styles

Design	Sliced	Reuse	Area [gates]	Delay [ns]	Synthesis time [min.]		
					loading	mapping	total
#1	-	on	989	25.0	2.2	7.0	10.6
		-	936	25.0		2.1	5.3
	yes	on	918	25.0	0.9	4.2	6.1
		-	917	25.0		2.4	4.2
#2	-	on	1911	25.0	54	8.6	66
		-	1852	25.0		5.6	63
	yes	on	1860	25.0	4.0	7.7	14.8
		-	1866	25.0		7.2	14.3
#3	-	on	4352	28.2	~16 hours	50	~18 hours
		-	4297	26.2		20	~17 hours
	yes	on	4228	26.3	24	39	97
		-	4399	25.3		28	113

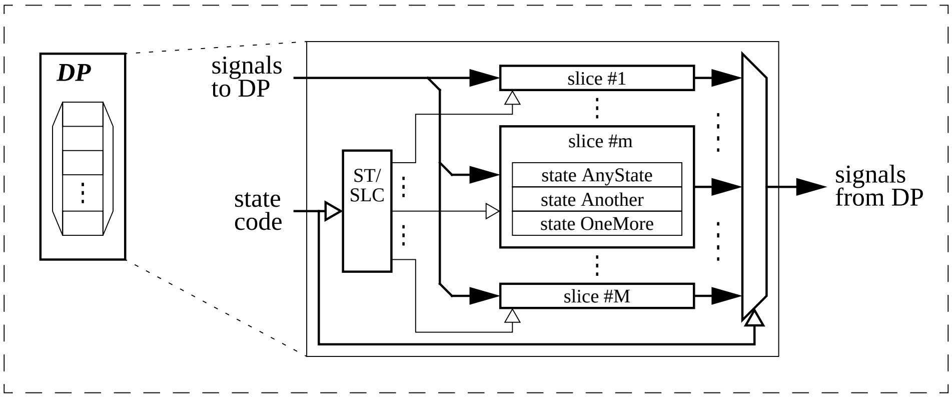


Figure 2.3. Data-path slicing

The quality of the result (area and delay) is more or less the same for all four ways. The main differences are in the loading and synthesis times. Although the reuse mode clearly affects the logic optimization phase (column “mapping”) the speedup of the whole synthesis process is insignificant compared against the synthesis times of sliced data-paths. The syntheses were performed on a Sun UltraSparc computer with 360 Mhz CPU and 1 Gb memory. The differences were even greater with older computers with smaller main memory - 40 hours versus 3 hours. It should be also noted that it took approximately 1 minute of CPU time to run all steps of xTractor when synthesizing the third design.

The slicing principles are simple (see Figure 2.3) - a slice encapsulates operations activated by some of the states and exactly one of the slices is active at any time. The output of the active slice is selected by a multiplexer. Additional encoder is used to deactivate unused slices. This encoder is especially useful when targeting low-power designs but it can be omitted, in principle. The current implementation of dividing states between slices is very simple - the first 4 states is grouped into the first slice, the next 4 states into the second slice, etc. This is based on a very straightforward assumption - operations in neighboring states are good candidates for reuse. More efficient state selection algorithms, which should take into account the actual closeness of operations are left for future research.

Figure 2.4. illustrates differences between RT level structures generated by a commercial HLS tool and xTractor. The boxes with shadowed background in the data-path represent bound functional units and registers. The main differences, aside the differences between the FSMs and in the number of functional units, can be listed as follows:

- multiplexers and comparators are kept in a single combinatorial block by xTractor to exploit logic optimization capabilities of the back-end tool (the boxes without shadowed back-

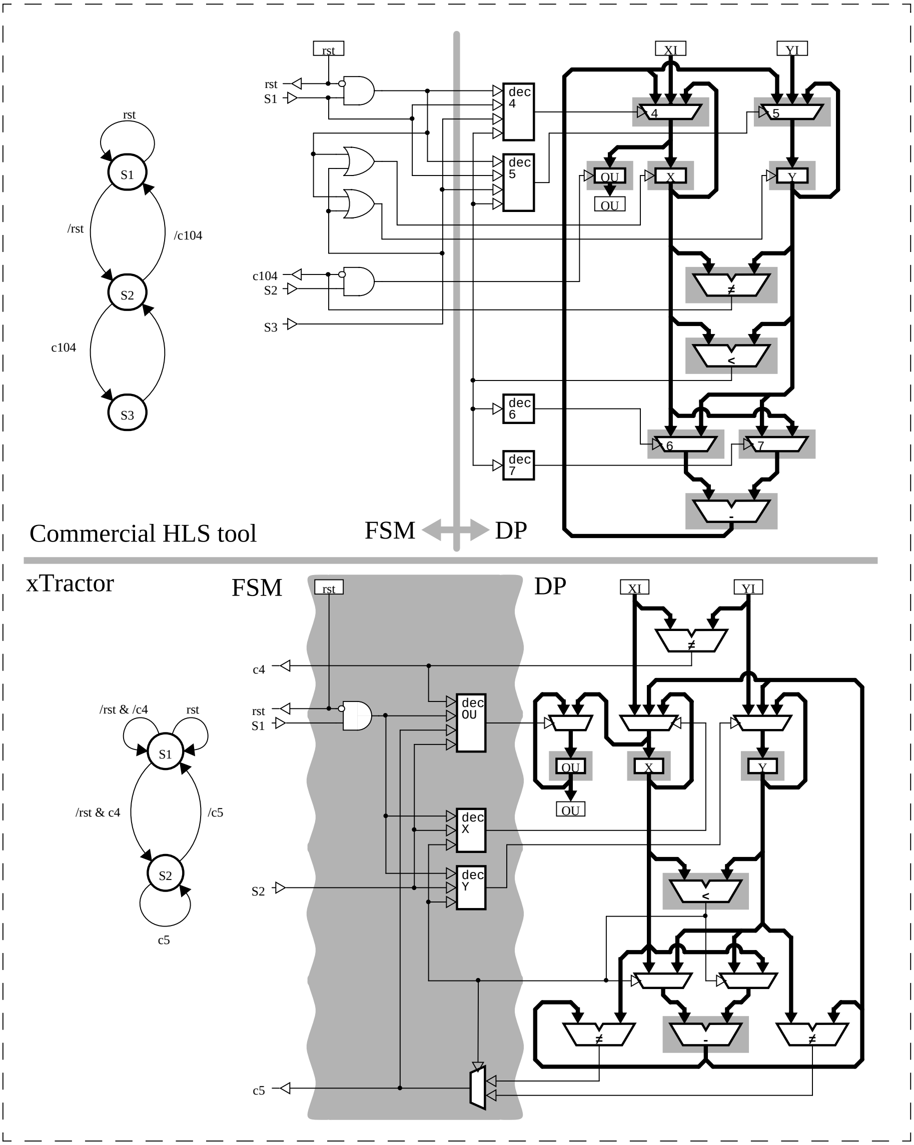


Figure 2.4. Comparison of generated RTL structures

ground); and

- it is hard to tell whether the decoders and random logic between FSM and DP belong to the FSM or to the DP, i.e. they can be optimized as a part of any of the sides (the shadowed are in the lower structure).

A synthesis script file for Synopsys Design Compiler (DC) can be generated together with the

RT level HDL code. The script allows to specify some design specific constraints, e.g. area and clock period, and to flatten design at different logic synthesis phases. [Syn92a], [Syn92b], [Syn92c], [Syn92d], [Syn92e]

Two of the main features of the xTractor's target architecture - operation chaining and the fact that multiplexers are treated as random logic - imply also the main drawbacks of the architecture. Reuse of functional units, which correspond to chained operations, creates a possibility for asynchronous feedbacks, called false loops; e.g. two reused arithmetic units are chained into a single clock step in different order in different states. False loops can be caused also by sharing also in control path. This situation can be generated actually by any binding algorithm. The false loops usually do not affect the actual hardware but complicate significantly synthesis tasks. The most affected is timing analysis. The possibility of generating false loops is very low in CMIST style applications, and never occurred in practice, because of the relatively small number of units which are reused. Nevertheless it is planned to incorporate into xTractor the detection and removal of false loops. A suitable methodology has been proposed in [SLH96]

2.4. Component programs

The tool set consists of eight component programs and of one shell. The interactive shell organizes the synthesis flow and executes the component tools in a defined order. The order and the options of programs can be modified to create different synthesis flow styles and saved as projects. The component tools can be executed also separately. Every tool, or a synthesis step, can be started from command line of the shell or from menus. The component tools are:

CDFG generator (cdfg_generator): Translates behavioral description of a design in a high level language into CDFG. Currently the only available input language is a subset of C, but translators from VHDL and SDL are under development.

Data-path handler (datapath_handler): This tool analyzes performance related properties of the CDFG and performs some simpler data-path optimizations - constant and variable propagation, and simplification of operations. The actual executable is "xfc_data". The tool is used also for analysis in the steps "Estimate delays", "Estimate area" and "Estimate performance" (commands "estimate_delays", "estimate_area" and "estimate_performance"); and in the synthesis flow in the steps "Propagate constants", "Simplify operations" and "Propagate constants (2nd)" (commands "propagate_constants", "simplify_operations" and "propagate_constants_memo" respectively).

Memory extractor (memory_extractor): Lists and/or maps arrays onto memories. The actual executable is "xfc_memxtr". The tool is used also for analysis in the step "Report memories" (command "report_memories"), and in the synthesis flow in the steps "List memories" and "Map / extract memories" (commands "list_memories" and "map_memories").

The tool generates also so-called memory mapping files where the mapping of arrays onto memories can be modified. A primitive built-in editor can be used for editing - step “Edit memory mapping” (command “edit_mapping”) in the synthesis flow or command “Edit mapping” in the “File” menu.

State marker (state_marker): Generates states marks while traversing the control flow of the CDFG. Various constraints - clock period, segment’s length, marking strategies, etc. - can be applied to guide the state marking. The actual executable is “xfc2fsm”. The tool is used also in the synthesis flow in the step “Mark states” (command “mark_states”).

SYNT-X state marker (state_marker_syntx): An implementation of the state marking strategy of the original CMIST approach. The tool is faster than the “state_marker” but may generate significantly inferior results. The actual executable is “xfc_syntx”.

Allocator / binder (allocator_binder): Allocates and binds functional units and registers. Interconnections (multiplexers) are allocated and bound together with related functional units and/or registers. The actual executable is “xfc_bind”. The tool is used also in the synthesis flow in the steps “Allocate & bind FUs” and “Allocate & bind registers” (commands “allocate_bind_fus” and “allocate_bind_regs” respectively).

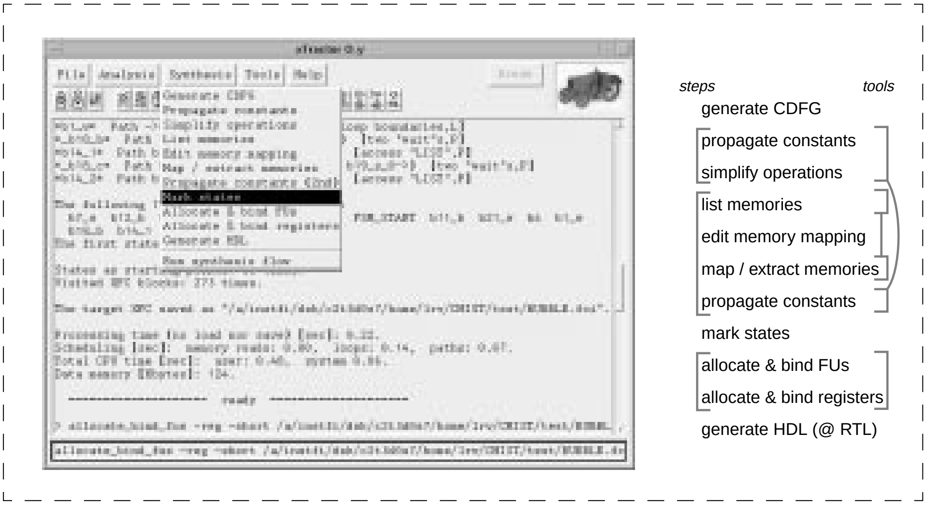
HDL generator (hdl_generator): Generates RT level synthesizable code in VHDL or Verilog. Synthesis script in dc-shell (for Synopsys DC) can be generated optionally. Different styles, selectable by options, target better exploitation of back-end synthesis tools. The actual executable is “xfc2hdl”. The tools is used also in the synthesis flow in the step “Generate HDL” (command “generate_hdl”).

Check CDFG (check_cdfg): Analyzes and reports some structural characteristics of the source CDFG. Allows to change buffering of ports and signals. The actual executable is “xfc_chk”. The same tool is accessible from the analysis menu.

Command line options of the component tools, if any, are listed in Appendix A.

2.5. Synthesis and analysis steps

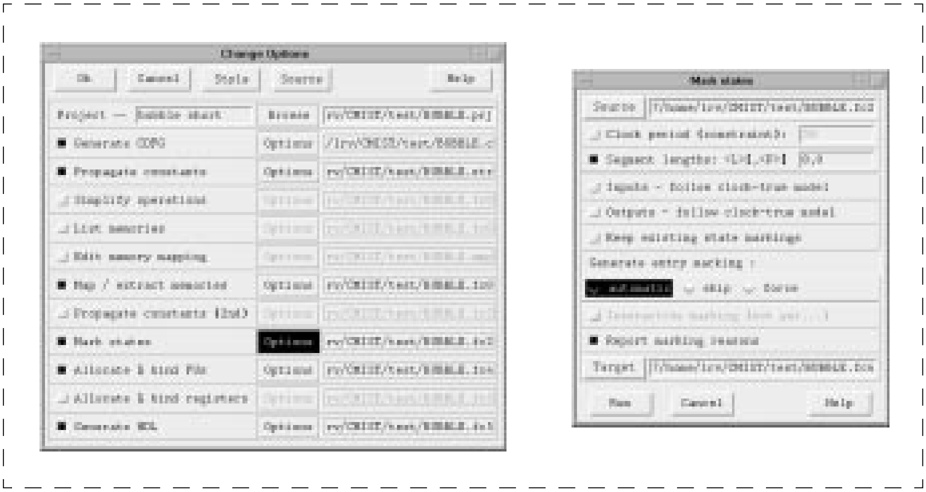
The synthesis flow, principle steps described in detail in section 2.2, can be executed step-by-step or as a single run. Figure 2.5. shows the main window of the tool set. The scroll-able text area contains reports from the component tools. This allows to inspect the results in detail. Synthesis steps can be executed by using menus or buttons, or a command can be typed into the shell’s command line. All available synthesis steps are listed in the right side of figure. Brackets on the left side of the list indicate which groups of steps can be skipped. Every group can be executed also as a single step.



```
Visited SPC blocks: 273 times.

Processing time (no load nor save) [sec]: 0.22.
Scheduling [sec]:  memory reads: 0.00,  loops: 0.14,  paths: 0.07.
Total CPU time [sec]:  user: 0.46,  system 0.06.
Data memory [Kbytes]: 124.

  -------------------  ready  -------------------

> allocate_bind_fus -reg -short /a/[illegible]/home/lrv/CHIST/test/BUBBLE.
```

Figure 2.5. xTractor - synthesis steps

The options of tools, and whether they are executed or not, can be set in corresponding windows. Figure 2.6. depicts the main option window and the options of the state marking phase. In the shown design flow the constant propagation and operation simplification steps are merged. The same is done with allocation and binding steps. The state marking options correspond to an unconstrained scheduling to get an AFAP schedule - no clock period nor segment look-ahead lengths have been defined.



Project — Bubble short

■ Generate CDFG
■ Propagate constants
□ Simplify operations
□ List memories
□ Edit memory mapping
■ Map / extract memories
□ Propagate constants (2nd)
■ Mark states
■ Allocate & bind FUs
□ Allocate & bind registers
■ Generate HDL

□ Clock period (constraint):
■ Segment lengths: <L>[,<F>] 0,0
□ Inputs - follow clock-true model
□ Outputs - follow clock-true model
□ Keep existing state markings
Generate entry marking :
● automatic ○ skip ○ force
□ Interactive marking (not yet...)
■ Report marking reasons

Figure 2.6. xTractor - design options

All commands are divided into four groups - component tools, analysis and estimation, synthesis flow, and supporting commands. A brief description of each group follows.

Component tools: Menu "Tools", contains the list of actual components programs described in section 2.4.

Analysis and estimation: Menu "Analysis". Commands in this group allow to analyze CDFG and to estimate area, delay and performance. Delay and area estimations of operations and units are based on technology dependent estimation models of functional units and storage elements.

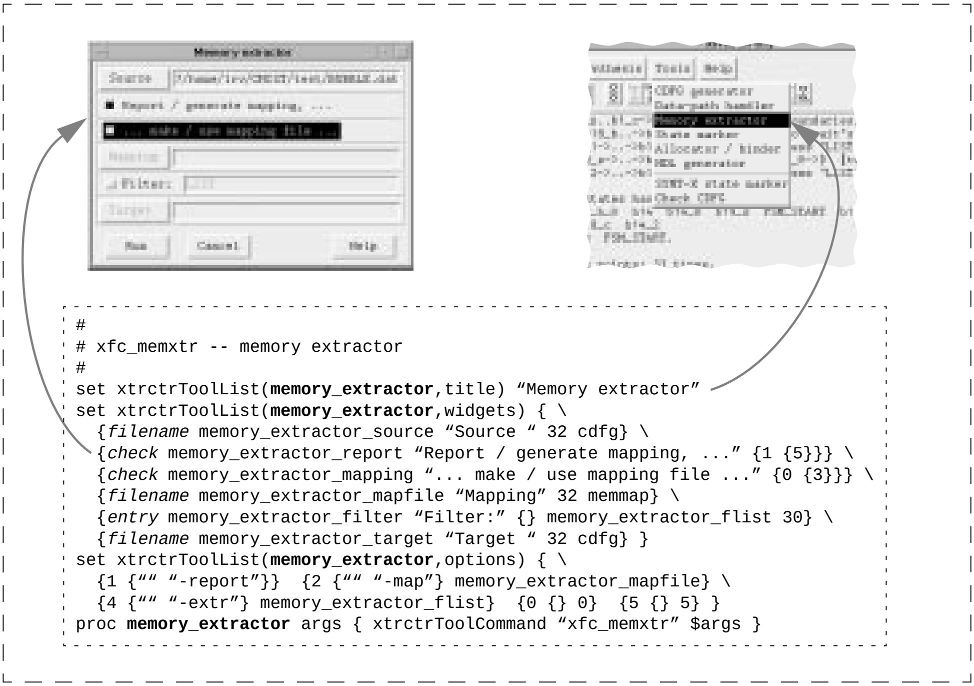
- *Estimate delays*: Estimates delays of CDFG blocks based on data dependencies. Finds also critical path(s) of state marked CDFGs.
- *Estimate area*: Estimates the total area by summing area estimations of CDFG blocks. Gives very approximate estimations.
- *Estimate performance*: Finds the number of different paths from entry to exit. Calculates also the number clock steps from entry to exit (loops are included once) of state marked CDFGs. The step can be very time consuming.
- *Report memories*: Lists sizes of unmapped memories (arrays). Interprets parameters of extracted memories from access signals (data/address buses). The interpretation is based on assumptions about extraction transformations and may be therefore erroneous.
- *Check CDFG*: Analyzes and reports some structural characteristics of the CDFG.

Synthesis flow: Menu "Synthesis". It is possible to skip some of the steps in the synthesis flow. There exist four different predefined flow styles - short and full flows with and without possibility to edit memory mapping. In the short flows data-path handling steps are merged into a single step, and allocation and binding of functional units and registers are performed together. The full flow consists of the following steps:

- Generate CDFG.
- Propagate constants.
- Simplify operations.
- List memories - skipped in the style without editing.
- Edit memory mapping - skipped in the style without editing.
- Map / extract memories.
- Propagate constants (2nd) - propagates constants used for memory addresses generation.
- Mark states.
- Allocate & bind FUs.
- Allocate & bind registers.
- Generate HDL.

Supporting commands: This group contains commands to load and save projects, to edit ASCII files (and CDFGs), etc. A special command allows to run Tcl/Tk scripts stored on disk.

The nature of Tcl/Tk allowed to build the shell of xTractor in a very flexible manner. It consists



```
#
# xfc_memxtr -- memory extractor
#
set xtrctrToolList(memory_extractor,title) “Memory extractor”
set xtrctrToolList(memory_extractor,widgets) { \
  {filename memory_extractor_source “Source “ 32 cdfg} \
  {check memory_extractor_report “Report / generate mapping, ...” {1 {5}}} \
  {check memory_extractor_mapping “... make / use mapping file ...” {0 {3}}} \
  {filename memory_extractor_mapfile “Mapping” 32 memmap} \
  {entry memory_extractor_filter “Filter:” {} memory_extractor_flist 30} \
  {filename memory_extractor_target “Target “ 32 cdfg} }
set xtrctrToolList(memory_extractor,options) { \
  {1 {““ “-report”}}  {2 {““ “-map”} memory_extractor_mapfile} \
  {4 {““ “-extr”} memory_extractor_flist}  {0 {} 0}  {5 {} 5} }
proc memory_extractor args { xtrctrToolCommand “xfc_memxtr” $args }
```

Figure 2.7. xTractor - internal structure

essentially of two layers. The lower functional layer consists of procedures to execute the component tools and to organize the interface between the tools and the shell. This layer also builds the main window and defines widgets to create the design flow and command groups. The second layer defines the actual menus, option windows, which options to use with which component tools, etc. This layer is defined as a set of data structures in Tcl/Tk stored into a single setup file. The lower layer tries to locate and to load three different setup files - global, user’s and local. Such layered buildup allows to customize the tool set not only for different users but also for different projects of the same user. Additional menus and/or commands can be defined in any of the setup files. Unnecessary commands can be blocked in a similar manner.

The declarations for component program “Memory extractor” are shown in Figure 2.7. as an example. Additional widgets, defined in the first layer, are shown in italics. The first command defines the title of the tool and that it belongs to the tool list. The second command defines the fields (widgets) of the option’s window. The second field, indicated by an arrow on the figure, defines a *check*able widget with label “Report / generate mapping, ...”. The third part of the field determines that the fifth field (“Target”) is blocked when the widget “has been checked” - no target is generated when in reporting mode. The third command defines how the argument list is built and the fourth command defines the command to be executed.

With the settings, shown in Figure 2.7., the shell will report parameters of unmapped arrays of the CDFG. The actual executed command is:

```
  xfc_memxtr -report /home/lrv/CMIST/test/BUBBLE.dat
```

Short descriptions are available for all steps and components of the tool step and can be listed by using “help” command.

2.6. Conclusion and future work

xTractor is a prototype tool set developed to test High-Level Synthesis methodology of CMIST style applications. The tool set consists of several smaller component programs, written in C/C++, to solve certain synthesis steps, and of an interactive shell, written in Tcl/Tk, around the component programs.

Since the tool set solves rather specific synthesis tasks several expansions are planned to add to target wider application areas. The main targets are control dominated designs with significant data path. The most important ones are listed below:

- translators to map different high level languages, e.g. VHDL, SDL and C/C++, into used internal representation (IRSYD);
- data and control flow transformations to reduce the impact of encoding style onto synthesis quality;
- automatization of data field pre-packing methodology described in chapter 4., and related memory mapping methodologies;
- elements of data flow oriented scheduling techniques for more efficient data flow synthesis;
- enhanced estimations for unified allocation and binding, plus false loop detection and removal to simplify back-end synthesis tasks.

The Tcl/Tk based shell allows to organize flexible use of other existing prototype tools for data and control flow transformations, for partitioning, for specific synthesis sub-tasks, etc. [Obe96], [Obe99], [ONi96], [ONi99]

3. IRSYD - Internal Representation for System Description

In this chapter an Internal Representation (IR), called IRSYD, is described. Motivations to develop such an IR are discussed. A synthesizable subset of IRSYD is used as an IR of the CMIST synthesis tool xTractor (see chapter 2).

3.1. Introduction

Specifications of large and complex systems require the combination of many description paradigms for different parts/features of a system. Structural modularity and hierarchy can be best described in a system description language like SDL, SpecCharts, VHDL or Esterel. Data processing parts may be best described by C/C++ or Matlab. Control dominated parts may be best described using communicating FSMs using StateCharts, VHDL or SDL processes. A different description language may be developed in future to more effectively describe communication and interfaces among sub-systems. Depending on the requirements any part of the system can be implemented in software, running on a CPU, as a dedicated or a reconfigurable hardware, or as a combination of both software and hardware.

Development of these complex systems requires not only integration of various types of specifications, but also integration of tools for partitioning descriptions into hardware and software. Needed are also tools for simulation and verification of the system; and tools for synthesis targeted to a set of specific architectures or implementation styles. This is an enormous task. Every research laboratory has developed some specific methodology to deal with these problems. An intermediate design representation or an Internal Representation (IR) is always an important element in these methodologies. Many times the IR is designed just keeping in mind the convenience of translation and requirements of one tool (generally synthesizer or simulator). As new tools are added to the system, it results in ad-hoc additions and modifications to IR. This not only makes the process of tool development more difficult, but also many times it is impossible to get efficient solutions because of the limitations of IR.

Therefore, design of an appropriate IR is of crucial importance to the effectiveness of a CAD environment. In this chapter, an Internal Representation for Hardware-Software codesign environment [ONi96] is described. The environment is planned to allow specification in a mix of many languages like SDL, Matlab, C etc. The tool is expected to integrate synthesis, partitioning, co-simulation, testing and formal verification, performance estimation, design optimization, and graphic visualization tools.

The basic idea of the methodology is described in Figure 3.1. The heart of the methodology is

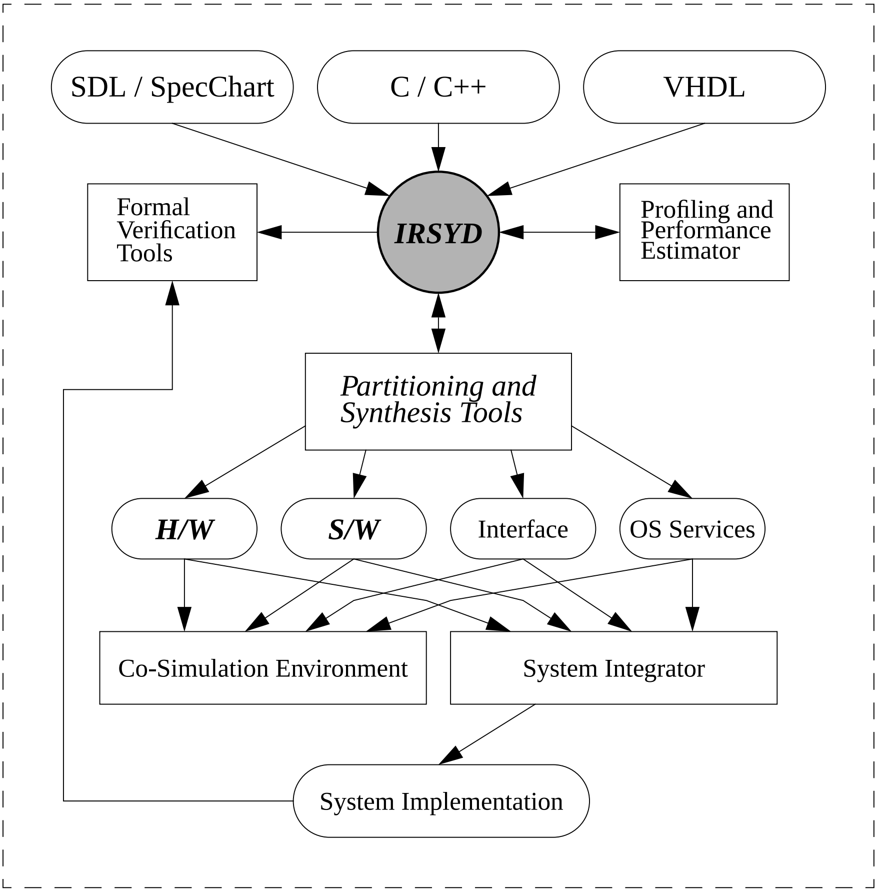

SDL / SpecChart
C / C++
VHDL
Formal Verification Tools
IRSYD
Profiling and Performance Estimator
Partitioning and Synthesis Tools
H/W
S/W
Interface
OS Services
Co-Simulation Environment
System Integrator
System Implementation

Figure 3.1. IRSYD's role in system design

an internal representation IRSYD (Internal Representation for SYstem Description). For an IR to preserve the semantics of the original specification in any language, it must have features to describe the following:

- static and dynamic structural and functional modularity and hierarchy;
- sequence, concurrency, synchronization and communication among various sub-systems;
- representation of abstract data types;
- mechanism for representing tool dependent information;
- mechanism for communication among various tools;
- reuse of design or behaviors.

IRSYD has been developed to meet these requirements. Unlike many other IRs, IRSYD is specifically targeted towards representation of heterogeneous systems with multiple front-end languages and to facilitate the integration of several design and validation tools. It forms the heart of the Hardware-Software codesign environment under development [ONi96], which will allow specification in a mix of several languages like SDL, Matlab, C/C++, etc. It will integrate synthesis, partitioning, co-simulation, testing and formal verification, performance estimation, design optimization, and graphic visualization tools. The specification of a system/subsystems could be in any of the languages for which a translator to IRSYD is available. Various synthesis, simulation, analysis and visualization tools can operate only on IRSYD and do not need to deal with the peculiarities of the original specification language.

The syntax of IRSYD was developed keeping in mind requirements for efficient parsing [ASU86]. It is stored on disk as ASCII text to make it independent of the order of bytes in CPU words and in memory. The order of tokens is defined in such a way that there is no need for lookahead parsing techniques. The lack of reserved words significantly simplifies mapping front-end languages into IRSYD. This makes it very uncomfortable for designers to read and/or modify it, of course, but the effectiveness of loading and storing is much more important for any IR.

IRSYD is also used as an IR of the prototype HLS tool set, presented in the thesis, which targets control and memory intensive applications (CMIST).

The existing internal representations are surveyed in section 3.2. Requirements for an IR are described in section 3.3. and the concepts of IRSYD are described in sections 3.4. through 3.6. Syntax of the IRSYD and its C++ class structure are described in Appendices B and C, respectively.

3.2. Existing Internal Representations

In this section, a brief introduction is given to some well-known IRs used by various research groups. Some of the listed IRs cover, in fact, more than one actual IR differing only in details. The strength and weakness of each of these IRs is also discussed.

3.2.1. Petri-nets and R-net

Petri-nets and their extensions have been effectively used to represent behavior of digital systems. Petri-nets can be used for representing sequential or parallel behavior. It can also be used for representation of behavioral hierarchy. Behavior of a system is modeled by a bipartite directed graph consisting of two types of nodes called *place* and *transition*. A place may or may not have a token. The state of the system is represented by the distribution of tokens in the system. The dynamic behavior of the system is described by flow of tokens in the system.

Various modifications and extensions of Petri-nets have been used as internal representations in high level and behavioral synthesis systems [KuBh79], [Pen87], [LKK92], [BDS95], [EKP98]. Computations in the system are described by actions associated with firing of a transition. It is not clear, however, how structure and structural hierarchy can be described using Petri-nets. Communication between parallel threads is only through shared variables.

3.2.2. SOLAR

SOLAR is intended as an intermediate design representation for control dominated mixed HW/SW systems [IsJe95], [JeOB95]. It is based on hierarchical, concurrent finite state machines. The principal building block is a StateTable. It allows the specification of hierarchical and communicating FSMs. In addition, three structures have been added to allow modular specifications. The DesignUnit construct is used to allow the structuring of a system description into a set of interacting sub-systems. The ChannelUnit construct allows the high-level specification of communication between concurrent sub-systems. The FunctionalUnitconstruct allows the high-level specification of shared operators between sequenced FSMs.

To allow the representation of a system in more than one way, the concept of a View is introduced. There are two supported views, the structural and the behavioral view, and several ViewTypes are defined for each view. For the structural view the ViewTypes “Interconnected systems” and “Communication systems” are defined; for the behavioral view the ViewTypes Communicating processes, Process level, and RT-level are defined.

3.2.3. Codesign Finite State Machines

Codesign Finite State Machine (CFSM) [CGJ94] is a network of communicating FSMs with a specific communication mechanism. The FSMs in the network are not synchronized with each other, i.e. the occurrence of their state changes is unrelated with each other and the time to compute the next state can vary and can be different for different FSMs, in fact it is unbound as long as the implementation is not known.

The communication mechanism between FSMs is based on timed events. An event is defined by a name, a value and a non-negative integer denoting the time of occurrence of the event. Events are transmitted in a send-and-forget MANNER and the sender does not expect or receive an acknowledgment. There is, however, an implicit storage element for each event type, which ensures that events remain available until e new event overrides the last one. There are two types of events: Trigger events can be used only once to cause a transition of a given CFSM, they implement the basic synchronization mechanism between CFSMs. Pure value events cannot directly trigger a transition but can be used to choose among several possibilities involving the same set of trigger events.

A timed trace of events is an ordered finite or infinite sequence of events with monotonously non-decreasing time of occurrences. This means that no events with the same name are simultaneous, i.e. no “communication channel” can carry two or more values at a time. A CFSM is defined by a pair of timed traces, one being a sequence of input events to the CFSM and the other being a sequence of output events produced by the CFSM as reaction to the input events.

The operational cycle of the CFSM goes through four phases: (1) idle; (2) detect input events; (3) transition, according to which events are present and match a transition relation element; (4) emit output events. Phases 1, 2, and 4 can take between 0 and infinite time, while phase 3 takes at least 1 time unit.

The model is very control oriented and does not support modeling of complex data types and data transformations efficiently. The communication mechanism is very specific. Although other communication mechanisms (blocking message passing, shared memory, etc.) can be modeled by this one and could be provided by the front end language, the question must be raised if it would allow for an efficient implementation in cases where this particular mechanism is not natural for the given application.

3.2.4. Program State Machines

The Program State Machine (PSM) [GVN94] model integrates a hierarchical, concurrent FSM with program language concepts. A PSM description consists of a hierarchy of program states. A program state is a generalization of a state in a traditional FSM. It represents a distinct mode of computation. At any given time, only a subset of program states will be active, i.e. actively carrying out their computations.

A program state can be composite or elementary. A composite program state consists of a set of program states, which operate either sequential or in parallel. An elementary program state is modeled as a sequential algorithm by means of an imperative language.

Transitions between sequential program states are modeled either as transition immediate (TI) edges or as transition on completion (TOC) edges. The TI edges are traversed as soon as the associated condition becomes true, no matter which sub-program states are active or where the computation is at that time. This allows model resets and exceptions. TOC edges are traversed when the program state has finished its activity.

Communication and synchronization between parallel program states is modeled by means of shared memory. No other means of communication like message passing, channels, etc. are provided.

3.2.5. CDFG

Many high level systems use a graph representation which can unify control flow and data flow in a system. Basic concepts required in such a representation are described in [DHG90], [GDW93], [GVN94]. A Control and Data Flow-Graph (CDFG) consists of two types of nodes, namely Control nodes and Data nodes. Control nodes are connected to each other to form a control flow graph. The control flow graph part of CDFG captures sequencing, conditional branching, parallelism and looping in the system behavior. Each node in control flow graph is linked to a data flow block. A data flow block consists of one or more sequences of data computations. Each sequence of data computations is represented by a directed acyclic graph (DAG) of data nodes. Each data node represents either an arithmetic/relational/logical operation or a read/write to a memory or a port. Variations and extensions of CDFG's have been used by many researchers.

3.2.6. Extended Syntax Graph (ESG)

COSYMA uses an internal representation called Extended Syntax Graph (ESG) [Cosima-www], which combines concepts of data flow graphs and syntax graphs. System is described as a directed acyclic graph describing a sequence of declarations, definitions and statements. ESG allows a mesh of pointers for efficient access of information for various tools like partitioner or estimator. The internal representation can be extended by adding more pointers. Data Flow information between operators is represented is by a second graph consisting of cross-linked blocks, called Basic Scheduling Blocks (BSBs). These BSBs contain local DFGs. Both these graphs share common operator nodes, thereby allowing easy transition between control flow and data flow and vice versa. It is not clear, whether there is any other mechanism, except shared variables, for communication between subsystems. ESG is intended for systems descriptions in C/C++.

3.2.7. Flow Chart

Flow charts [Lee76], [FeHi93] have been used for many years in system design as an informal description of algorithms. A flow chart is an informal graphical description of an algorithm built as a directed graph. Usually four types of nodes - start, stop, operation and decision - are used but depending on problem area different other types can be used. The flow charts have been widely used to describe algorithms in early phases of designing both software and hardware, mainly because of their capability to give a good overview of the algorithm but also to simplify checking the logic. There exist also some special expansions of flow chart for hardware design - sequential machine flow charts, behavior finite state machines and algorithmic state machine charts, to name some of them.

3.2.8. Extended Flow Chart (XFC)

The basic limitation of all FSM based IRs (SOLAR, CFSM) is its control oriented nature and its reliance on states which makes it unnatural for data flow and event dominated systems.

A control flow oriented design representation called Extended Flow Chart (XFC) [EHK96] was developed to be used as an internal representation for synthesis of CMIST (Control and Memory Intensive System) applications [EKS96]. XFC can be easily obtained from behavioral descriptions in languages like VHDL or Verilog. XFC is similar to a flow chart like representation called Graph Scheme of Algorithm (GSA) [BaSk86] in the sense that states are marked on the graph edges. This is different from other flow chart like notations, say, ASM charts [Wak94] in which each operation node represents a distinct state. Marking states on the edges allows the state boundaries to be specified in a flexible manner. The BFSM notation [TWL95] has similar flexibility of specifying the state boundaries but like ASM and GSA, it does not allow describing data path operations directly.

IRSYD is a further development of XFC extended to handle hierarchy, both structural and behavioral, different data types, including abstract types, and different communication mechanism.

3.3. Features required in a unifying Internal Representation

An IR, required for supporting multiple front languages, must be capable of representing all the concepts of these languages. Since support of SDL, VHDL, C and Matlab is envisaged, the unifying IR will be required to represent the following:

- structure and structural hierarchy to describe a large system as a composition of subsystems;
- behavior and functional hierarchy to represent a complex behavior in terms of composition of functions;
- data and computation on data;
- communication among subsystems; and
- timing and timing constraints.

Besides these features, the IR should ensure that no useful semantic information in the source language, which may be useful for its synthesis or analysis is lost. IR should also have information in a format, which is suitable for various tools like simulation, analysis, partitioning and synthesis. This may cause redundancy of information in IR. Another important feature of such an IR should be that it should have mechanisms to reuse designs at any level of behavioral and structural hierarchy.

There are two distinct approaches for building such an IR. The approaches can be called as Union of Concepts and Primitive Atomic Concepts.

Union of Concepts. In this approach, the unifying IR will contain mechanisms for direct representation of concepts of all the front-end languages. In such an IR, if the two languages have identical semantics for a particular concept then they can share a common internal representation for that concept. Otherwise, the IR will provide different mechanisms for representing their

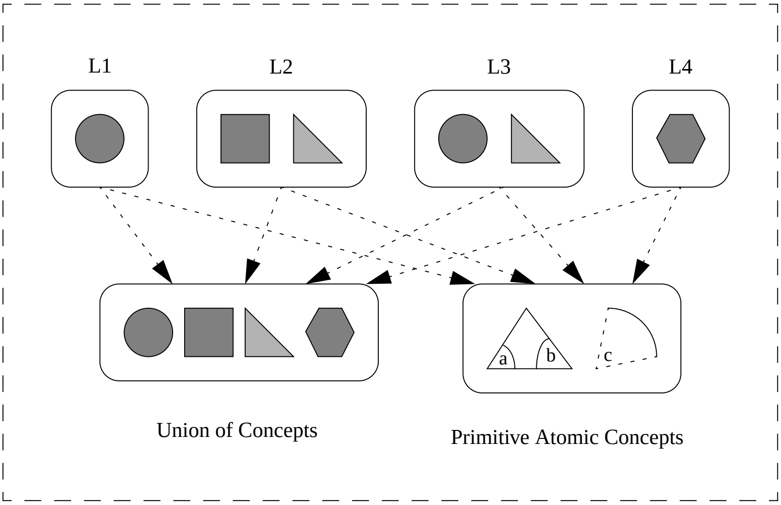


Figure 3.2. Union of Concepts versus Primitive Atomic Concepts

concepts. For example, VHDL allows inter-process communication through shared variables and ports, and SDL provides inter-process communication through message passing and through limited sharing of variables. An IR based on Union of Concepts will provide:

- representation and semantics for communication through shared variables;
- representation and semantics for ports;
- representation and semantics for SDL specific message passing; and
- representation and semantics for SDL specific sharing of variables.

Primitive Atomic Concepts. In this approach, the unifying IR will represent concepts of all the front-end languages in terms of a set of primitive atomic concepts. This set has to be "complete" in the sense that any language concept can be represented using a composition of concepts from this set. For example, the communication concepts of both SDL and VHDL could be represented through shared variables with mechanisms to specify access procedures and access rights to this variable.

Figure 3.2. illustrates the distinction between two techniques graphically. Any polygon may be constructed using triangles. Edges of various radius and angle may be used to construct any circle, etc.

Table 3.1 describes the advantages and disadvantages of the two approaches. It should be noted that the major disadvantage with the Primitive Atomic Concepts approach is the loss of source language specific information. When looking, for example, at a specific component of the IR based on primitive atomic concepts, it may not be possible to find the source language and the construct of the language from where it was generated. This information may be very useful to a tool to produce efficient implementation.

The Union of Concepts, at the other hand, is never stable because it must be extended whenever a new front-end language is added. IRSYD is based on a mix of the two approaches. It uses a set of Primitive Concepts (not necessarily atomic or minimal) covering the concepts of all the languages of concern. To avoid loss of information a special attribute mechanism is provided, which allows to recover the full information content of the front-end description. Attributes, which can be assigned to any of IRSYD's elements, are also useful for exchanging information between various tools. The attribute mechanism and other supporting concepts, used in IRSYD are described in section 3.6.

Table 3.1. Union of Concepts versus Primitive Atomic Concepts

	Union of Concepts	**Primitive Atomic Concepts**
Memory Requirement	Small	Large
Translation to IR	Easy	Relatively Difficult
Information loss	"Lossless"	"Lossy"
Extensibility and stability of IR	Difficult to extend and unstable	Stable
Simulation and verification tool design	Difficult	Relatively easy
Synthesis tool design	Difficult	Relatively easy
Reverse translation / back annotation	Easy	Difficult

3.4. IRSYD concepts

IRSYD is based on XFC (eXtended Flow Chart) [EKS96], an internal representation used in the high level synthesis of Control and Memory Intensive SysTems (CMIST) [HSE95] for its semantics. The basic idea in IRSYD (as in XFC) is to have a representation in which it should be possible to express control flow and data flow in an integrated graph representation. Concepts for modeling hierarchical systems and systems modeled as a set of concurrent processes have been also updated/added. In this section, the basic concepts of IRSYD are described. The dynamic behavior of IRSYD is described in the next section through its execution model.

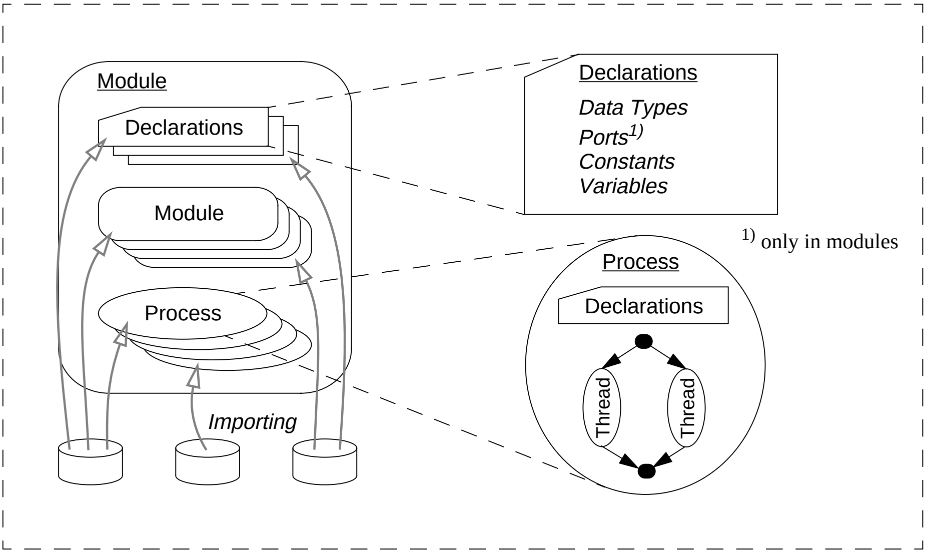


Figure 3.3. Hierarchy of IRSYD elements

3.4.1. Module

Module is the basic unit of hierarchy in IRSYD (see Figure 3.3). A module itself consists of sub-modules, *declarations* and *processes*. Declarations include module's interface description (*ports*), and definitions of *data-types*, *constants* and shared *variables*. All declarations in a module are visible in its sub-modules but not vice versa. The processes are the leaf level elements in the structural hierarchy where the actual computation takes place. A process consists of local declarations (types, constants and variables), visible only locally. It has also one or more execution threads, which represent the control flow. A detailed description of processes and execution threads is given in the following section about the execution model.

At the top level, a system is initially defined as an empty module (top-module) into which all lower hierarchical parts are imported. Import is allowed into all modules independent of their location in the hierarchy. This feature allows to build a module's body from many different sources, e.g. the parts may be located in different files.

Processes are created and their execution is started immediately on activation of the enclosing module. A module can be activated during the execution of a process. This allows creating modules and processes dynamically and recursively. The deactivation (killing) of a module and process is implicit. A process is killed when one of its threads reaches an exit node indicating the end of execution. A module is deactivated if all the processes within it have completed computation. It can be seen that this set of elements are sufficient to represent structural and functional hierarchy described in the languages (VHDL, SDL, C/C++, Verilog, etc.) of interest.

3.4.2. Sequential computation

Sequential Computation is represented by sequence of flow nodes of a thread. A special node type - operation node - is associated with a set of assignment statements. The order of these statements implicitly implies the data dependencies in the computation.

All expressions are built in prefix notation; i.e. operation is the first token, followed by parameters. This makes all operations to look like function calls in effect. An addition of two variables, for instance, is represented as a function call with two parameters. The main reason is efficient parsing - it is always known how many operands must follow when an operation has been detected. Call of user defined functions, which may have any number of parameters, have an extra operand just after the function name (operation label) - the number of parameters. Examples of representing an addition and a function call in C/C++ are depicted in Table 3.2.

Table 3.2. C/C++ expressions mapped into IRSYD

C/C++	IRSYD
`( a + b )`	`+ a b`
`atan2 ( x, y )`	`atan2 2 x y`

Function is a special case of module to allow also behavioral hierarchy. There exist two types of functions in IRSYD - open and closed. Access of global variables, i.e. read or write, is allowed by open functions. A function is called closed if it does not access a global variable for evaluation; that is, it does not have side effects. These functions can access only data elements, which are described inside the function itself (parameters belong to the function). This means also that a pointer can be accessed but not data, pointed at by that pointer, if the data exists outside the function. Normal, i.e. open, functions do not have this restriction. Hierarchical buildup of the internal structure (used by simulator/synthesizer) must provide corresponding information. Functions are always used following the “call-by-value” model, i.e. parameters are copied and then passed to the function.

3.4.3. Parallel computation

There exist two levels of parallelism in IRSYD, namely concurrent processes and parallel threads. A system can be composed of one or more IRSYD modules. An IRSYD module describes one or more processes. Each process can contain one or more parallel threads, as illustrated by Figure 3.4. Note that IRSYD modules, processes, and threads describe the static structure of the system, but only processes and threads are concepts for describing dynamic, par-

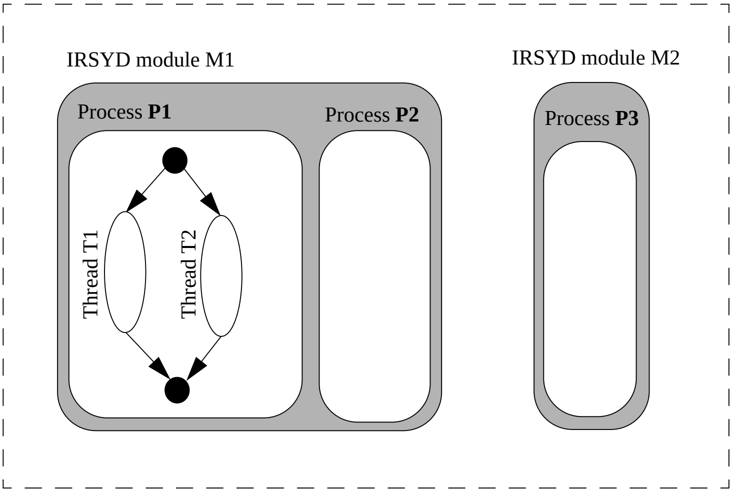


Figure 3.4. The relation between IRSYD modules, processes and threads

allel activity.

- Concurrent processes are asynchronous. Synchronization between concurrent processes is achieved with events and send/receive operations. Only variables explicitly declared as shared variables are common to two concurrent processes. All processes are created dynamically. A process is killed when the exit node is executed. One “main” process is created by an external mechanism when the system’s execution is started. One or more concurrent processes are created with a create node, which takes a name of an IRSYD module as parameter. As many processes are created as are described in the named IRSYD module.
- Parallel execution threads are synchronized by *split* and *join* operations, which start and stop threads. All the variables and their declarations are shared by all execution threads within that process.

3.4.4. Communication

IRSYD provides three distinct ways to represent communication between concurrent processes and parallel threads, namely ports, shared variables, and messages. Communication primitives of various front-end languages of interest can be easily mapped onto them.

Shared variables: A variable, which is visible to two processes, can be used as a shared variable for communication among them. Same scope rules are applicable to shared variables during communication, as they are applicable during computation. Shared variables can also be used by parallel threads to communicate with each other.

Ports: Input/output ports allow for communication between concurrent processes in different IRSYD modules. The ports are established upon activation of a new module.

Since a process can use the same port to communicate with two or more other processes, the port can be a means for symmetric communication between any number of processes across IRSYD module boundaries. It is essentially a shared variable. To preserve access rights and usage information from the front-end language, ports can be of the following types: *in port*, *out port*, *in-out port*. An extra flag, *copy*, can be added for buffering purposes when activating a module. The first three merely indicate how the port is used by the modules. The copy type means that the port is copied into a local variable when the module is activated.

Message passing: Message passing is realized with *send* and *receive* operations. Both operations name a channel they want to send to or receive from. A channel is a virtual, infinite buffer, which is identified by a unique identifier, which is an integer. There is no explicit creation or destruction of a channel. Channels are created when they are first accessed and never destroyed. Since both sender and receiver must identify the channel, a message can be read by any process without restriction.

Send and receive operations are used to access the data in the buffer of the channel. In order to represent various flavors of message passing mechanisms in different front-end languages, send and receive are very powerful and flexible operations in IRSYD. Both send and receive can be indexed to access arbitrary locations in the buffer, and both come in two variations, a destructive and a non-destructive one.

The index of receive operation counts the location in the buffer from the head of the queue. Thus, an index of 0 denotes the first element in the queue, which is presumably the oldest element. The index of send counts from the tail of the queue. Thus, an index of 0 denotes the last element in the queue, which presumably is the youngest element. If all send and receive operations accesses the channel always with index 0, the behavior of the buffer is a traditional FIFO.

Receive can either remove the accessed element from the queue or copy it and leave it there. Send can either overwrite the accessed element or insert the new element before the addressed element. If send uses insert mode with index 0 it appends the new element at the tail of the queue.

This flexible concept allows for instance the modeling of message passing mechanisms found in Erlang and SDL, where message acceptance depends on the content of the message.

3.4.5. Representation of data

Data is represented in IRSYD by means of several primitive and composition data types.

Built-in data types:

- *boolean*: values TRUE & FALSE;
- *string*: unbounded array of characters;
- *integer*: unbounded (from minus infinity to plus infinity);
- *signed* & *unsigned*: bit-true representation of integer values;
- *pointer*: "address" of a variable;
- *pointer-onto-module* (pointer-onto-function): this is a reference to an function represented in IRSYD and can be used for describing higher order functions;

Set types: A set type defines a set of labels. IRSYD does define neither any mapping of set types onto integers nor any built-in operations on these types like comparisons. If a language, which makes such assumptions about set types, e.g. enumerated type in C/C++ or VHDL, is translated into IRSYD, the translator has to explicitly generate these mappings and corresponding operations.

Structured data types:

- *bounded array*: any set type or integer range can be used for indexing an array.
- *unbounded array*: any data type can be used as index, which allows to represent an associative memory.
- *records*: structures with elements of any data type.

Built-in operators: Built-in operators are only defined for the types *boolean*, *string* and *integer*. For booleans IRSYD defines the logic operations *AND*, *OR* and *NOT*. For strings it defines the string concatenation and for integers the arithmetic operations *PLUS*, *MINUS*, *MULTIPLY* and *DIVIDE* based on the number theory in mathematics. Note that these might not be identical to the arithmetic operations in a programming or hardware description language, because these languages often assume a finite range of integers and a particular representation.

All other operators for built-in and user-defined types must explicitly be provided by the IRSYD generation program.

Function and operator overloading is not allowed - it is an internal representation and front-end tools should map them onto corresponding functions (sub-modules).

Type conversions should be performed by using special functions (sub-modules).

3.5. Execution model

The dynamic behavior of IRSYD, i.e. the semantics of its description, is defined by the execution model.

As mentioned earlier, the actual computation in the system takes place in the active processes.

Control flow in a process is described by Flow Chart [Lee76] like representation, based on an earlier similar representation XFC (Extended Flow Chart [EKS96]).

A process is a directed cyclic graph. The nodes give control blocks and the edges represent the precedence relationship among these nodes. To represent various situations, there are seven types of nodes, namely *entry*, *exit*, *operation*, *condition*, *split*, *join* and *activate*.

- Every process has exactly one *entry* node, which mark the beginning of the control flow in that process. This node is executed first when the process is activated.
- A process may have none, one or many *exit* nodes. These nodes indicate the completion of computations described in the process. On reaching this node the process is deactivated (or killed).
- All computational activity of a process is encapsulated into *operation* nodes.
- *Condition* nodes are used for two-way branching based on value of a boolean variable, or multi-way branching based on a value of a variable of set type.
- *Split* node splits execution thread into two or more parallel threads, which share the same set of variables. These created threads can be later merged into a single thread using *join* node.
- *Activate* nodes are used to activate sub-systems described as IRSYD modules activating processes in corresponding module.

The execution time of any node is assumed to be zero. An event can be associated with any edge in the graph. An event is an expression, using variables within its scope, giving boolean result. An event is first evaluated when execution arrives at the edge to which it is associated, and execution continues if the result is true. If not, evaluation is repeated when any of the variables changes until the result is true. A lazy-evaluation is assumed to simplify underlying evaluation mechanisms of tools. Events can be used for synchronization between different threads in a process, or different processes within a system or a process with the external world. This mechanism allows us to model time.

The evaluation order of events triggered at the same time is not defined. A system should be described in a manner that the order of execution of simultaneous events does not affect the overall behavior of the system. This is caused by the underlying execution engine, which can use different models to select the event to be evaluated.

3.5.1. Activation and deactivation of a module

Execution of a system starts with activation of the top-module. Activation of a module means that all constants, variables, and ports, which are declared in the module, are created and allocated according to their declaration and all sub-modules and processes are activated. A module can be activated also during the execution an *activate* node (function call), allowing to create modules and processes dynamically and recursively. Activation of a process is performed by creating all constants and variables, described in the process, and executing its entry node.

The deactivation of modules and processes is implicit. A process is deactivated when any of its *exit* nodes is executed. All constants and variables are deallocated and all sub-modules, activated by the process, are deactivated. A module is deactivated when all its processes have been deactivated, or deactivated by the execution engine. All constants and shared variables are deallocated. Ports are not touched because they belong to modules on higher hierarchical levels. A module is deactivated when the process, by which it was activated, has been deactivated. In that case all its sub-modules and processes will be deactivated. Deallocation of variables is the main reason why all sub-modules must be deactivated.

There is no special built-in mechanism to inform higher level processes that a process/module has been deactivated. A simple mechanism is to wait for an event, which is triggered by the process before it is exiting (see Figure 3.5).

Note that IRSYD modules, processes, and threads describe the static structure of the system, but only processes and threads are concepts to describe dynamic parallel activity.

3.5.2. Node execution

Execution of a node depends on its type:

- *Entry* node is used to inform the execution engine where to start the execution. There is exactly one *entry* node per process.
- Execution of an *exit* node informs the execution engine that corresponding process must be deactivated. A process may have none, one or many *exit* nodes.
- *Operation* nodes are the only ones, which can modify variables and trigger events; i.e. they encapsulate all computational activity of a process. Statements in a node are executed in strictly sequential order.
- *Condition* nodes analyze values of a variable and inform the execution engine, which of the alternative threads to be executed. They are used for two-way branching based on the value of a boolean variable, or multi-way branching based on a value of a set type variable.
- *Split* nodes inform the execution engine that execution continues in parallel threads.
- Execution of a *join* node informs the execution engine that all entering threads must synchronize at this point - execution can be continued only when the execution of all threads arrives at the node.
- Execution of an *active* node informs the engine that the corresponding module and its processes must be activated. At the same time execution of the current thread must be continued. It can be said that the execution of an activated node is functionally equivalent to the execution of a split node where one of the threads corresponds to the activating thread and the rest corresponds to the processes of the module to be activated. The difference is that activation is performed on a lower hierarchy level.

After execution of a node, its edges to successor nodes are analyzed. Conditional expressions of events associated with the edges are evaluated. An event is first evaluated when execution arrives at the edge to which it is associated, and execution continues if the result is true. If not, evaluation is repeated when any of the variables changes until the result is true. A lazy-evaluation is assumed to simplify underlying evaluation mechanisms of tools. Events can be used for synchronization between different threads in a process, or different processes within a system or a process with the external world. This mechanism allows us to model time. The evaluation order of events triggered at the same time is not defined.

It is important to note that the execution engine must evaluate all events where execution threads have been stalled. The simplest way is to organize execution in two main steps:

- Execute all nodes in threads, which have not been stalled. The execution order of nodes is not important. One thread can be executed until a conditional expression associated with an event has to be evaluated and then the next thread is executed.
- Evaluate all events, which wait for evaluation. If the result is true, mark this thread as active, otherwise as stalled. The order of evaluation of events is not significant.

To avoid system level deadlocks, when all active threads have been stalled and none of the event evaluations returns true, a special built-in function can be defined to detect such a situation. The function *idle* returns true when all active threads have been stalled. A simplified time model of VHDL, described in the next section, is based on the usage of function *idle*.

The identification of active threads by the execution engine must not be visible for executed modules, and is not defined. Process identifiers, which are for instance required by operating systems and for message passing can be organized in many different ways. One of the simplest ways is to use global shared variables. The values to these variables are assigned in the very beginning of a just created process (or thread). Assigning undefined values to these variables just before exit node can be used to inform about deactivation of the process.

An example of modeling a function call, i.e. execution of the caller must be stopped until the sub-module has been deactivated, can be seen in Figure 3.5., where variable *rdy* is used to indicate that called function has finished. Assigning *true* value to that variable triggers an event on the thread where execution of the caller was paused. The graphical representation of IRSYD objects is based on Flow Charts. Node types are differentiated for readability.

However, the precise way, how this is modeled in IRSYD will depend on the front-end language and must be defined by conventions (see section 3.6.).

3.5.3. Exceptions and errors

Errors in the IRSYD description do not cause necessarily the execution to be stopped immediately. IRSYD only defines that expressions, which cannot complete correctly, must return a special value ‘*nil*’. All expressions, which get a value ‘nil’ as argument, must also evaluate to ‘nil’.

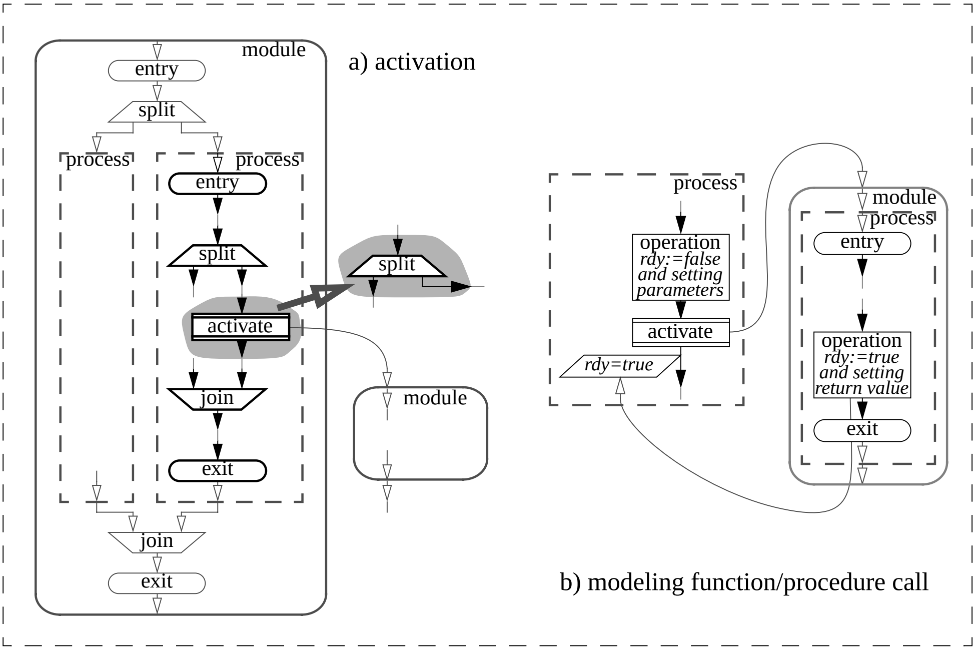


Figure 3.5. IRSYD execution model

In this way, an error propagates and it is the responsibility of the execution engine to check for errors and react in an appropriate way.

Exceptions are part of the system modeled in IRSYD such as resets. They can be modeled in IRSYD without additional constructs. Assume for instance a typical reset behavior, where an event causes the current thread of execution to be stopped and some variables are reset to their default values while others should keep their current values. Two levels of modules can be defined for this case. The top level defines variables which must “survive” the reset sequence - they are visible as ports or shared variables for the lower level module. The control flow activates the next level and waits for deactivation of that level. After deactivation, the next level is activated again immediately, thus effectively restarting the algorithm. The lower level describes behavior of the algorithm with one addition: just after the entry node the thread is split into two parallel paths. The first one implements the algorithm. The second path waits for the reset event and exits, resulting in deactivation of that level.

This might seem too complicated way to model such a simple concept like a reset. However, since IRSYD is an intermediate representation, and a reset concept, which might be part of the front-end language, can be captured with above described mechanism together with a set of attributes which interpret this mechanism.

3.5.4. Modeling time

There is no special concept of time in IRSYD to avoid language dependent behavior. Events are used to organize execution order of modules/threads, making it possible to model time based on this simple mechanism. Conventions can be used to standardize different time models. As an example, a simplified model of time for VHDL is described in section 3.6.

3.6. Supporting constructions and concepts

In this section, various supporting constructions and concepts are described.

3.6.1. Attributes

Tools need general means to add non-functional information to IRSYD, which in turn can be used by other tools. To this end, IRSYD has an attribute mechanism, which is both simple and flexible. Each node can be annotated with an attribute list. Each attribute consists of a name and a value; the value is a list consisting of 0, 1, or more elements. There is one dedicated place where global attributes for the entire system are stored, which is the entry node of the system. The attributes are used both to avoid loss of information when mapping front-end language constructions onto IRSYD, and to pass information between various tools.

3.6.2. Re-use and library subsystems

For fast development of large and complex systems, it is necessary to reuse modules, which have been previously designed. IRSYD provides mechanism to include external modules (in the IRSYD form). The modules to be used may be available in separate files or in an organized manner. There is no concept of libraries in the IRSYD itself, to keep it simple and to avoid possible problems of mapping libraries onto different file systems.

A module (system) using an external module must import declarations to indicate the part of the system, which should be loaded from the disc. It is the responsibility of the designer and tools to make sure that the imported module (sub-module) is used properly. It is the responsibility of a tool to access the physical file(s) corresponding to the external module and build the required information.

3.6.3. Foreign language interface

IRSYD provides an interface to models written in a foreign language. The purpose of the interface could be reuse of earlier design, generation of test data, analysis or display of response to

```
mod OpenFile {
  @foreign(“lang=C” “name=fopen” “def=stdio.h” “lib=libc.a”)
  port {
    fid: out ‘int;
    name mode: in ‘str;
} }
```

Figure 3.6. An example of a foreign language interface

test data, and to provide access to the OS depending utilities (file system accesses, special purpose library functions, etc.). From the IRSYD point of view all foreign language functions/modules are described as IRSYD modules, i.e. interface to the module is defined in terms of ports of suitable types. Because an IRSYD module lacks return value, typical for a function, it should be modeled as an extra output port.

The syntax of IRSYD defines only how the interface should be described. Mapping details are not part of the IRSYD and depend on the underlying interface modules of tools and attributes can be used to synchronize the mapping.

The execution of a function/module described in a foreign language is initiated by activating the corresponding IRSYD module. Depending on how the activation is described in terms of IRSYD primitives, execution of the activating module can be stopped, until the activated module has finished; or can be continued in parallel with the activated module. Refreshing of ports and shared variables uses the same mechanism as it is used between any two IRSYD modules and should be provided by the tools, i.e. the underlying interface modules must ensure correct propagation of new values.

In the example in Figure 3.6., a library function is mapped onto IRSYD module. The function is *fopen* and its C-interface is used.

3.6.4. Conventions

There are two contradicting requirements on IRSYD:

- It should be simple without superfluous concepts and constructs;
- It should be able to represent all concepts of the front-end languages without loss of information. A loss of information can already occur if a front-end language primitive is expressed with an assembly of primitives in IRSYD, although the input-output behavior is equivalent.

For instance, if a send/receive communication is modeled in IRSYD by means of a shared variable with associated control, we might loose information because a synthesis tool may not be able to infer the original send/receive mechanism from the IRSYD model. The tool may consequently generate an inferior implementation compared to one, which may result directly from the send/receive primitive.

To resolve this conflict a two level modeling approach is proposed. The lower level is based on the core IRSYD syntax and semantics, which is attempted to be simple and powerful. In the second level, conventions define how various front-end language concepts are modeled in IRSYD. For each front-end language, a set of conventions must be provided which defines how concepts and primitives of the concerned language are represented in IRSYD. Attributes play a major role in the conventions because they can be used to tie an assembly of IRSYD primitives together and associate them with a higher level semantics.

Every convention has a unique name and whenever an IRSYD model follows a particular convention, this must be stated explicitly in an attribute. The attribute name is the name of the convention and its value holds convention specific additional information.

3.6.5. Modeling of time

The following description is an example of a convention about how to model time in IRSYD. For the whole system, a common clock, representing current time, is used. There exist two basic actions, which must be supported by the time model:

- *Delay* - Execution of a thread should be paused for a certain period of the system time. This means that the thread is paused until the current time will be equal to the current time at the moment of pausing, plus value of the desired delay.
- *Updating system clock* - Because there are idle periods between events the system clock must be updated independently from the modeled system. A situation where all active threads are waiting for an event and at least one of them is waiting for a new value of the current time indicates that the system clock must be updated.

The simplest detection is based on function *idle*, which indicates that all active threads have been paused. The proposed convention consists of one type declaration (*time*), two variable declarations (*now* and *TimeUnit*), one module description (*UpdateSystemClock*) and two function descriptions to represent operations with type time (*time_add* and *ge_time*, not shown here), all shown in Figure 3.7.

As an example, a simplified time model of VHDL, together with modeling of statement “***clk <= not clk after 10 ns;***“, and interactions with system, are shown in Figure 3.8. It uses function *idle*, which indicates that all active threads have been stalled. Data type *time* is described as a record consisting of two fields (to emulate floating-point numbers), and the functions *time_add()* and *time_ge()* perform addition and comparison, respectively. The variable *now* corresponds to the current time and *TimeUnit* to the smallest interval. Functions *not()* and *idle()* are built-in, vari-

- Type *time* - record of two integers, one for exponent and another for mantissa:

```
type { time: ‘rec { m e: ‘int; }; }
```

- Shared variables *now* and *TimeUnit*, both of type time:

```
shvar { now TimeUnit: time; }
```

- Module *UpdateSystemClock* - one of the top-most modules, i.e. always active:

```
mdl UpdateSystemClock {
  prc update {
    const { zero: time := (0,0); }
    entry begin reset;
    op reset [ now zero ] incr ‘( ‘idle );
    op incr [ now time_add 2 now TimeUnit ] incr ‘( ‘idle );
  }
}
```

Figure 3.7. Modeling system time in IRSYD

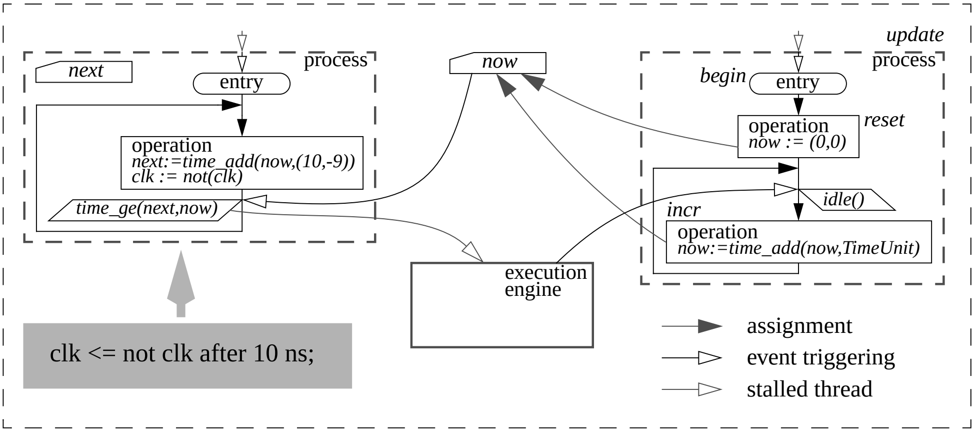


Figure 3.8. Modeling VHDL clock in IRSYD

able *next* (of type *time*) is used to evaluate when the stopped thread should continue.

3.6.6. Tools perspective of IRSYD

As discussed in the first section, an internal representation should fully capture the information in the initial specification so that it is possible to synthesize an implementation, which is semantically equivalent to the specification. An internal representation should present different views to tools so that these tools can extract or add required information to the internal representation conveniently and efficiently. This implies that internal representation may have the same information about the system repeated in different forms. Internal representation should also have a mechanism for communication among various tools. In summary, an internal representation should have the following features:

- Semantic equivalence with initial specification.
- Possibility of multiple views for convenience and efficiency of tools.
- Mechanisms for communication among tools.

IRSYD has all these features.

Language translators. These tools are IRSYD generators from initial specification in different languages. The tools have the responsibility to fill IRSYD with the information for the convenience of other tools. IRSYD has means to represent the concepts in all the system specification languages, programming languages and hardware description languages. As discussed in section 3.4., IRSYD can represent parallel and sequential computations. It can represent structural and behavioral hierarchy; it has mechanisms for communication between subsystems; it can represent timing and timing constraints; it can represent asynchronous as well as synchronous behavior. These features make it an internal representation independent of language.

However, a given behavior modeled in a specification language can be represented, in principle, in many different ways in IRSYD. IRSYD is designed in a way, that it can represent all concepts of the specification language. This might not always be possible in a straightforward and natural way, without losing information and without making design decisions. In order to provide the IRSYD model with the full information content of the front-end language model, conventions must be defined and used by the translation tools in a way that synthesis and analysis tools can extract the front-end language modeling concepts efficiently.

HL synthesizer. HLS requires both data flow and control flow representation of the system for carrying out synthesis steps. It also require interface to pre-synthesized (or a library) of modules for generating a structural netlist of the design. The resultant netlist can also be easily represented in IRSYD since IRSYD has the concepts to represent hierarchical interconnections of blocks. It is quite easy to extract and synthesize control part from IRSYD. [EHK96], [EKS96], [EKS96b], [EKH97]

HW/SW partitioner. The partitioner needs to identify design parts that can be implemented in a particular technology, independent of other parts. For this, the interfaces of these parts must be explicit. A main issue is the granularity of partitioning. In IRSYD, the potential subjects for HW/SW partitioning are parallel threads, processes, and IRSYD modules.

Simulator. An IRSYD native simulator will require that the operational semantic is precisely specified. In order to simulate IRSYD modules together with other simulators (e.g. test bench) a foreign language interface to C and VHDL must exist.

3.7. Conclusion

The need and importance of a unified internal representation for design of heterogeneous systems has been discussed in this chapter. The features required in such an internal representation have been identified. An internal representation (IRSYD), which has these features, has been proposed and designed. An important new concept in IRSYD is the mechanism of retaining source language specific information through special attributes called conventions.

The full definition of IRSYD in BNF is given in Appendix B. Some details about IRSYD usage, not discussed in this chapter because of their secondary importance, are discussed in details in [EKJ97]. Various solutions to model communication in IRSYD are described in [ONi97] and [MJE99].

IRSYD is implemented in C++ as a class library. A translator from SDL to IRSYD has been defined [SKE98]. Future work will cover VHDL to IRSYD translator, converting hardware synthesis tools ([EKH97], [EKS96], [EKS96b], [EKH97]) from XFC to IRSYD. In addition, visualization, simulation, estimation and partitioning tools are planned.

4. Pre-packing of Data Fields in Memory Synthesis

This chapter describes a methodology to exploit data transfer locality in high-level memory mapping. The methodology is especially well suited for data transfer intensive (DTI) telecommunication applications, which are characterized by dynamically allocated data structures (DADS).

4.1. Introduction

High-level programming and hardware description languages are well suited to describe telecommunication applications, especially data-transfer intensive, at the system and behavioral levels. One of the characteristics of such applications is usage of abstract data types, which allow effectively to encapsulate hierarchy of data elements, and hide low level details in higher levels. The use of arrays of records and/or structures, depending on the description language, is very typical for DTI telecommunication application. These arrays can be combined in different ways into different physical memories. Every combination - mapping - affects many characteristics of the whole design space - size, performance, power consumption, etc. This depends mostly on the relationships between different fields of data records, data and control dependencies. To find the best or a satisfying, at least, mapping is not a trivial task.

Since memory has always been a dominant factor in DSP ASICs the researchers in this community where one of the first to address the problems related to memory as an implementation component [VCV89], [VLV95]. Storage requirements of control ASICs are often fulfilled by registers and rarely require large on chip memories on the same scale as the DSP ASICs. Control applications that require large storage, like in protocol processing, were in the past implemented in software. These days, protocol processing being a bottleneck in network traffic [Tan96] has led to research effort in implementing such functionality in hardware. The complexity of protocol processing applications, its history of being implemented in software and demands on productivity motivate use of dynamically allocated data structures while specifying and modeling such applications. Matisse [SYM98] is a design environment under development at IMEC that addresses implementation of such structures and other issues related to the implementation of protocol processing functionality.

DADS are defined on the basis of functional and logical coherence for readability. Retaining such coherence while organizing the DADS physically in the memory does not optimize the required storage, nor does it optimize the required bandwidth. These optimization goals are addressed by analysis of the dependencies between accesses and using them as the basis for packing elements of DADS into a single memory word. The pre-packing step has the following benefits:

- minimizes the total number of data-transfers;
- minimizes the total storage by reducing bit-width wastage;
- reduces search space complexity for the rest of the physical memory management flow;
- reduces addressing complexity.

In this chapter, the methodology of pre-packing of elements of DADS is presented. First, an overview of related works is given. A brief overview of Matisse design environment and place of the pre-packing in it is presented. Next, each step of the pre-packing phase is described. Finally, the results of applying approach to some ATM cell processing applications are presented. A more detailed description of applying the pre-packing on a real life design is discussed in the section 7.2.

4.2. Related work

Almost all published techniques for dealing with the allocation of storage units have been scalar-oriented and they employ a *scheduling-directed* view (see e.g. [KuPa87], [BMB88], [GRV90], [SSP92]) where the control steps of production/consumption for each individual signal are determined beforehand. This applies also for memory/register estimation techniques (see e.g. [KuPa87], [GDW93], [DePa96] and their references). This strategy is mainly due to the fact that applications targeted in conventional high-level synthesis contain a relatively small number of signals (at most of the order of magnitude 10^3). The control/data-flow graphs addressed are mainly composed of potentially conditional updates of scalar signals. Therefore, as the major goal is typically the minimization of the number of registers for storing scalars, the scheduling-driven strategy is well fitted to solve a *register allocation* problem, rather than a *background memory allocation* problem.

The above techniques present serious shortcomings for most real-time protocol processing applications for several reasons. First, scheduling *must* precede (background) memory management in the conventional high-level synthesis systems. However, by switching the order, the dominant background memory cost can be reduced further [VCV89], [VCV91] and the freedom for data-path allocation and scheduling remains almost unaffected [CGD94]. Furthermore, within the scalar view, many examples are untractable because of the huge size of the ILP formulations.

Exceptions to this focus on scalar applications have been initiated by early work at IMEC and Philips. Phideo [VLV95] at Philips is oriented to periodic stream based processing of video signals. At IMEC, since 1988 extensive research has been performed in the direction of this array-oriented custom memory organization management. For the protocol processing target domain, the most relevant work is the one on automated memory allocation and assignment *before* the scheduling or the procedural ordering are fully fixed [Bal95], [SWC97]. This has lead to a significantly extended search space for memory organization solutions, which is effectively explored in the HIMALAIA tool. In addition, the important storage cycle budget distribution step should be noted, where the bandwidth requirements and the balancing of the available cycle budget over the different memory accesses is determined [WCD96]. A complete methodology for custom background memory management (or data transfer and storage exploration as it is called) has been proposed in the ATOMIUM script [CWD98]. In addition, extensions to the network component (e.g. ATM) application domain have been developed in the physical memory management (PMM) stage of the Matisse project [SWC97].

Recently also several other approaches for specific tasks in memory management oriented to non-scalar signals in multi-dimensional signal processing have been published. This includes the MeSA [RGC94] approach at U.C. Irvine focusing on static memory allocation, and at CMU where strategies to mapping arrays in an algorithm on horizontally and vertically partitioned dedicated memory organizations have been proposed too [ScTh97]. The latter approach uses horizontal concatenation of arrays, similar to the proposed pre-packing of elements of DADS, but does it at the same time with scheduling thus significantly increasing the complexity of the optimization task. Therefore, the exploration scope of the technique is heavily restricted. Moreover, the focus of the pre-packing approach includes also dynamically allocated data types with sets of records.

4.3. Memory organization exploration environment

Matisse is a design flow intended for system exploration and synthesis of embedded systems characterized by dynamic data storage and intensive data transfer. The four main steps of the Matisse design flow are as follows (see also Figure 4.1). [WCD96], [MCJ96], [MCJ98], [SWC97], [SYM98]

- During *dynamic memory management refinement* the actual structures of dynamic data types and virtual memories is decided.
- The goal of *task concurrency management* is to meet the overall real-time requirements imposed to the application being designed.
- *Physical memory management* (PMM) aims to synthesize area and power efficient distributed memory architectures and memory management units.
- *High-level address optimization* phase maps address expressions of the algorithm onto optimal address generating architectures.

For custom targets, conventional high-level synthesis tools and methods are used at the back-end. These are complemented with system-level exploration aiming to reduce the time-multiplexed unit cost.

Usually, the available cycle budget is insufficient to perform all memory accesses sequentially. Therefore, some of accesses must be performed simultaneously. The PMM decides the order of

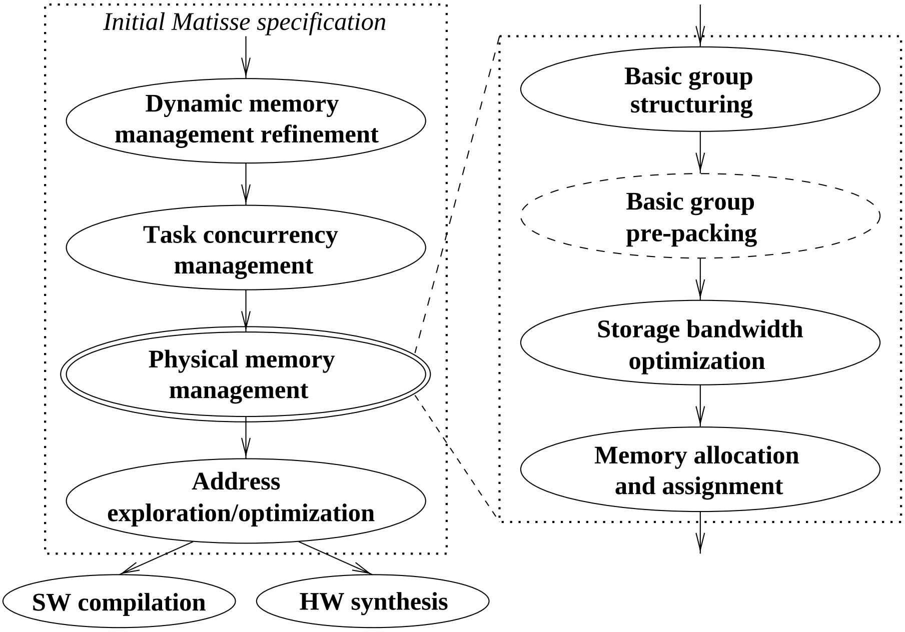

Initial Matisse specification
Dynamic memory management refinement
Task concurrency management
Physical memory management
Address exploration/optimization
SW compilation
HW synthesis
Basic group structuring
Basic group pre-packing
Storage bandwidth optimization
Memory allocation and assignment

Figure 4.1. Matisse design flow

the accesses and distribution signals over physical memories. Register and other scalar oriented assignment problems, e.g. allocation and binding in high-level synthesis, are well known [GDW93]. However, to deal with realistic applications, the *memory management* task should assign groups of scalars to memories instead of individual scalars. These non-overlapping groups of scalars are called basic-groups (BG). This is done in such a way that for every memory access, it is known at compile time which basic-group is being accessed. All three major steps of PMM, listed below, work with basic-groups.

- *Basic-group structuring* explores different virtual memory segment layout. This step heavily influences the optimal memory architecture as it affects the number of memory accesses and the word width.
- *Storage bandwidth optimization* analyses dependencies between memory accesses and generates a partial order of memory transfers that leads to final schedules with minimal number of access conflicts. [WCD96]
- *Memory allocation and assignment* phase determines the most efficient memory configuration in terms of area and power. The step takes the memory access conflict graph, generated by the previous step, and applies a weighted conflict graph coloring algorithm. [SWC97]

The proposed pre-packing of fields of dynamic data structures works also with the basic-groups, and is executed before the storage bandwidth optimization. Pre-packing mergers multiple memory accesses but may also introduce extra accesses. The details of the pre-packing are discussed in the next sections. It should be added that, in principle, the PMM phases from pre-packing to allocation and assignment can be iterated many times to explore potential solutions.

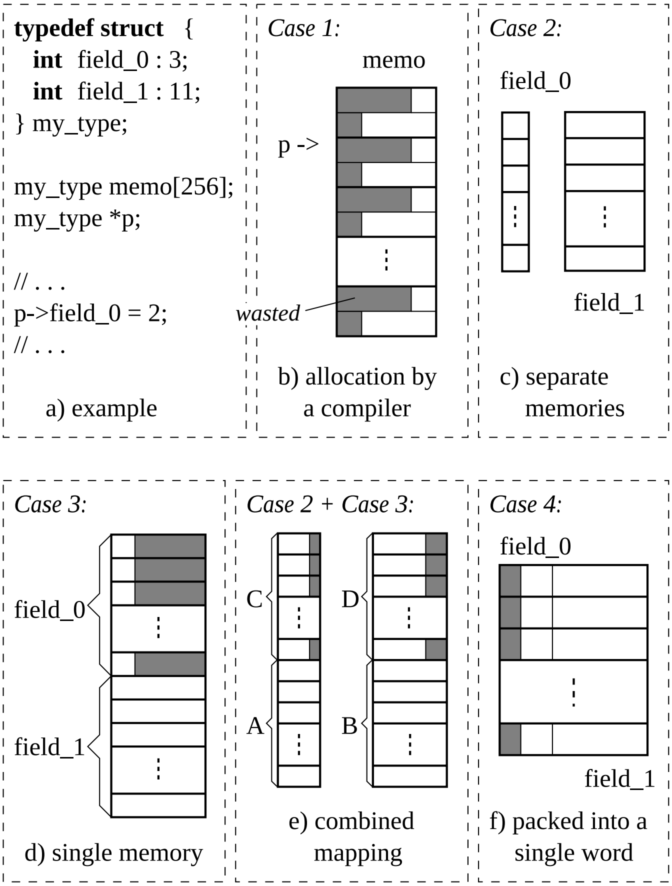
4.4. Pre-packing of basic-groups - problem definition and overall approach

Dynamically allocated data structures require often rather large tables to store them - millions of bits is nothing exceptional. Since it is cheaper to use available on-chip memory cores than to custom design them; it is beneficial to use the memory area and bus-widths as effectively as possible. Also, large memories, especially when multiple tables are allocated into a single physical memory, are sometimes too big to be implemented as on-chip memories. Usage of external memories with even less limited amounts of available configurations puts even higher stress onto efficient usage of memory area. Accesses to external memories are also more power hungry.

The main goal of pre-packing of fields of data-structures, basic-groups in the context of Matisse environment, is to reduce the number of memory accesses thus reducing primarily the power consumption. Additionally it also helps to reduce the bit-with wastage of memories and simplifies succeeding synthesis/optimization steps.

The way, how different fields of a data structure are mapped onto physical memories affects significantly the size of the memories and the number of accesses needed to fetch/store the data. The number of accesses directly affects the overall performance and power consumption. Let us consider a simple example where the structure `my_type` consists of two fields `field_0` and `field_1` with width 3 and 11 bits respectively (see Figure 4.2.a). An array `memo` is defined to be of `my_type`. Only two basic-groups can be identified in this example - arrays corresponding to `field_0` and `field_1` - since no data is available about access addresses. Consider the following different cases of mapping this simple structure to memory and its consequences:

- Case 1: Software compilers would typically map the array in such a way that an array element (structure) is accessed by a pointer and different fields within the element by different offsets. This is shown in Figure 4.2.b, where unused bits are shadowed. Similar solutions in hardware would be undesirable, mostly because of the wasted memory area.
- Case 2: Mapping fields onto different physical memories, as shown in Figure 4.2.c, reduces the wasted area but potentially increases the address calculation cost, both in area and power. The main advantage, of course, is higher performance due to the simultaneous memory accesses without using multiport memories.
- Case 3: Another solution is to map both arrays of fields onto the same memory - a case somewhat similar to the software solution except the difference in the offset (Figure 4.2.d). The drawback is that no simultaneous accesses are allowed, or multiport memories should be used. Combining cases 2 and 3 allows to make trade-offs between performance and bus width - the case 2 allows simultaneous accesses but requires wider buses [SWC97], [WCD96]. An exam-



```
typedef struct   {
  int  field_0 : 3;
  int  field_1 : 11;
} my_type;

my_type memo[256];
my_type *p;

// . . .
p->field_0 = 2;
// . . .
```

a) example

b) allocation by a compiler

c) separate memories

d) single memory

e) combined mapping

f) packed into a single word

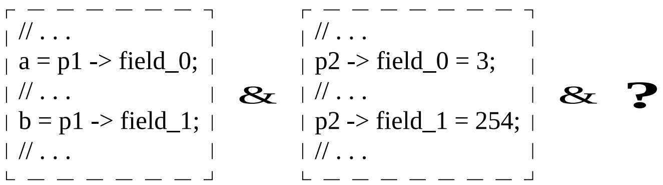
Figure 4.2. Different mapping strategies

ple of mapping four arrays into two physical memories is shown in Figure 4.2.e.

- Case 4: The fourth way is, of course, to pack two or more fields of the same structure into a single memory word (Figure 4.2.f). The advantages are potential reduction in the number of memory accesses and uniform width of memory words. The downside of such a solution is that updating a single field requires two accesses - one to fetch the unchanged fields and another to store back the modified data. This may be acceptable if the total number of accesses is reduced.

An intuitive approach, based on the experiences of designers, is rather simple in its nature - two similar accesses (read or write) can be merged when (Figure 4.3):

- the same base address (`p1` or `p2`) is used, i.e. the fields belong to the same data element; and
- the candidate fields are accessed in a basic block, indicating data transfer locality. In Figure



```
// . . .
a = p1 -> field_0;
// . . .
b = p1 -> field_1;
// . . .
```

&

```
// . . .
p2 -> field_0 = 3;
// . . .
p2 -> field_1 = 254;
// . . .
```

& ?

Figure 4.3. Candidates for intuitive packing

4.3., two instances exist where `field_0` and `field_1` are accessed in a basic block, the more such instances; the greater would be the motivation to pack `field_0` and `field_1` in a single memory word.

It is possible to analyze accesses and pack basic-groups manually, and to reduce the number of data transfers and/or memory waste. However, it is difficult and time consuming to analyze trade-offs. Manual analysis together with manual code modification is also a source of potential errors. A systematic approach is needed which takes into account that there exist situations where packing is beneficial even when both fields are not accessed together. An additional read, for instance, is insignificant compared to the total reduction in the number of memory accesses. The basic costs are area and power within given real-time constraints. For data storage and transfer related issues, these costs are directly derivable from the memory size and the number of memory accesses [CWD98].

Based on the discussion above the main steps of the proposed approach can be outlined. As the first step, the pairs of memory accesses where packing of corresponding data fields affects the total number of accesses should be collected. Various graph optimization based approaches can be used for the analysis of collected pairs. The analysis phase is mapped onto hierarchical clustering of compatibility graph of basic-groups. The three mains steps, described in details in the following sections, are:

- collect dependency cases between all pairs of relevant (with respect to locality of the data-transfer) memory accesses;
- build a compatibility graph based on the characteristics of the dependencies; and
- decide packing, i.e. merge/split fields.

4.5. Collecting data dependencies

A simplified CDFG is used for dependency analysis since only information relevant for the data-transfer is needed. The only nodes of interest of the pruned graph are read and write operations. Dependencies between these operations are also kept.

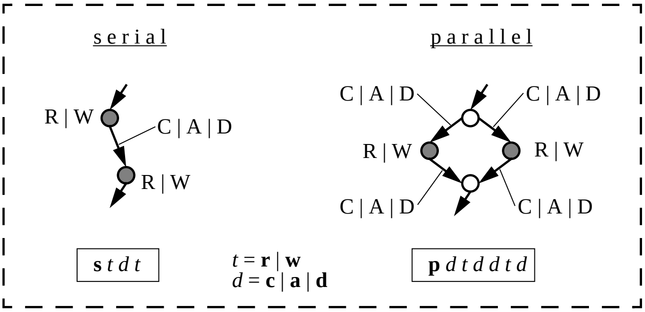


Figure 4.4. Dependency types

Pairs of the memory accesses (read or write) can be classified as one of the two main types:

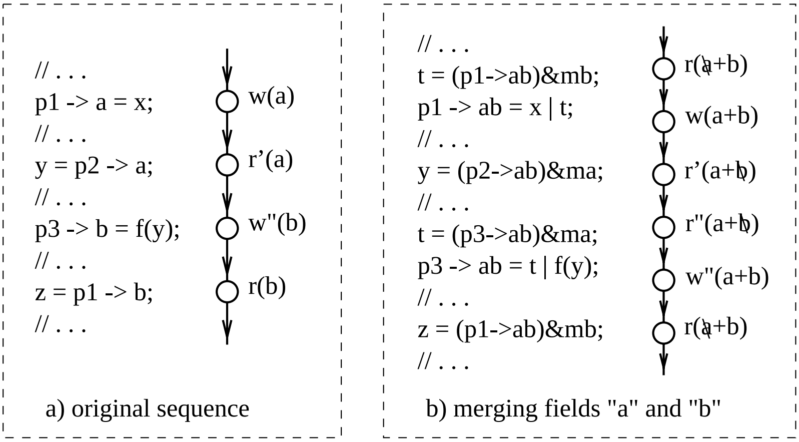
- there is a sequential dependency between accesses, i.e. one of the accesses directly depends on the second one;
- parallel accesses - both accesses share a common predecessor and a common successor.

The predecessor can be the start of the process and the successor can be the end of the process (return from the process). The types of dependencies are control, data and address dependencies:

- *control dependency* - the decision to execute the path or not is dependent on the preceding operation;
- *address dependency* - the address used by the access uses data from the preceding operation; and
- *data dependency* - the data used or generated by the access is dependent on the preceding operation.

A read operation of a field, for instance, is data dependent on the preceding write operation when the data is written onto the same address (of the same data field). Figure 4.4 depicts all potential sequential and parallel dependencies. The pair "*R | W*" denotes read and write operations and the triple "*C* | *A* | *D*" the type of the dependency - control, address or data. The notation for sequential dependency - "***s*** t_p *d* t_s" - describes types of the predecessor (t_p) and successor (t_s) nodes and type of the dependency (*d*) between them. The parallel dependency - "***p*** d_p *t* d_s d_p *t* d_s" - describes in a similar manner two accesses (d_p *t* d_s). For instance, "***s*** *r d w*" describes a sequential dependency where the write operation is data dependent on the preceding read operation.

The following well-motivated assumptions are used in the analysis:



```
// . . .
p1 -> a = x;
// . . .
y = p2 -> a;
// . . .
p3 -> b = f(y);
// . . .
z = p1 -> b;
// . . .
```

a) original sequence

```
// . . .
t = (p1->ab)&mb;
p1 -> ab = x | t;
// . . .
y = (p2->ab)&ma;
// . . .
t = (p3->ab)&ma;
p3 -> ab = t | f(y);
// . . .
z = (p1->ab)&mb;
// . . .
```

b) merging fields "a" and "b"

Figure 4.5. Long dependency chain

- accesses to the same basic-group are ignored;
- a single execution thread is assumed, i.e. all accesses can be statically ordered on a relative time axis;
- only pairs of accesses to the same data structure and depending on the same pointer (base address) are analyzed;
- only static dependencies are analyzed to avoid the need for profiling data (which may be misleading); and
- a simplified CDFG can be used - since only memory accesses are of interest, the only nodes in the CDFG are read and write operations.

It is also assumed that only access paths with length one are analyzed, i.e. the immediate predecessor nodes are taken into account only. The main reason is that the long dependency chains are covered by shorter ones. Another reason is that while neighboring accesses can be potentially scheduled into the same clock step, the same can not be said about the accesses on long chains. In Figure 4.5., an example of a long sequential dependency, before and after packing, is depicted - reading from field “*b*” may depend on the writing into field ‘*a*’ (type “**s w d r**” in Figure 4.4) only when there is a longer dependency chain. ‘*r(a)*’, ‘*r'(a)*’ and ‘*r"(a)*’ mark loading the same field using different pointers; ‘*r(a̸+b)*’ marks loading packed fields ‘*a*’ and ‘*b*’ while discarding the field ‘*a*’; and ‘*ma*’ and ‘*mb*’ are masks to extract single original fields.

The total number of all possible combinations of sequential accesses is 12 - there are two possibilities for the first access type and two possibilities for the second access type, and three dependency types between these two accesses: 2*2*3=12. The total number of parallel dependencies, 324, is found in the same manner - two nodes of two access types each and four edges of three dependency types each: 2*2*3*3*3*3=324.

The number of possible dependency cases is much smaller when eliminating infeasible and equivalent combinations:

- dependency types of succeeding edges of parallel accesses do not affect pre-packing thus giving 36 (2*2*3*3) cases only;
- a read operation can be data-dependent only on the writing into the same BG (see assumptions of the analysis above), giving 5 sequential and 21 parallel possible cases in total;
- some of the combinations are symmetrical - there is no distinguishing between left and right operations of parallel accesses - which leaves 18 parallel cases;
- address dependencies are conflicting with the assumption that the same base address should be used;
- assumption that only one control thread is active at any time eliminates parallel cases where only one of the branches has control dependency.

The remaining eight dependency cases are listed in the Table 4.1. Two special sequential cases - read/write is address dependent on the previous read - are added. The reason is that they illustrate results of pre-packing when the second access is effectively independent on the previous access. These two cases, “single-read” and “single-write”, explain also why the write operations should be treated differently from the read operations. The last elimination, based on the assumption that only one active control thread exists, refines also the parallel cases:

Table 4.1. Dependency cases

Case			Old accesses		New accesses		Change
1	read-after-read	srcr	*r(f1); r(f2);*	2	*r(f1+f2);*	1	-1
2	write-after-read	sr-w	*r(f1); w(f2);*	2	*r(f1+~~f2~~); w(f1+f2);*	2	0
3	read-or-read	pcr-cr-	*r(f1); ⁄ r(f2);*	1	*r(f1+~~f2~~); ⁄ r(~~f1~~+f2);*	1	0
4	read-read	par-ar-	*r(f1), r(f2);*	2	*r(f1+f2);*	1	-1
5	read-or-write	pcr-cw-	*r(f1); ⁄ w(f2);*	1	*r(f1+~~f2~~); ⁄ r(f1+~~f2~~); w(f1+f2);*	1⁄2	0 ⁄ +1
6	read-write	par-aw-	*r(f1), w(f2);*	2	*r(f1+~~f2~~); w(f1+f2);*	2	0
7	write-or-write	pcw-cw-	*w(f1); ⁄ w(f2);*	1	*r(~~f1~~+f2); w(f1+f2); ⁄* *r(f1+~~f2~~); w(f1+f2);*	2	+1
8	write-write	p-w--w-	*w(f1), w(f2);*	2	*w(f1+f2);*	1	-1
I	single-read	srar	*r'(f1); r(f2);*	2	*r'(f1+~~f2~~); r(~~f1~~+f2);*	2	0
II	single-write	sraw	*r'(f1); w(f2);*	2	*r'(f1+~~f2~~); r(f1+~~f2~~); w(f1+f2);*	3	+1

- that two parallel accesses are control dependent on their common predecessor when they are mutually exclusive, i.e. they belong to the different control threads (basic blocks); and
- the parallel accesses are data/address dependent on their common predecessor when they belong to the same control thread (basic block).

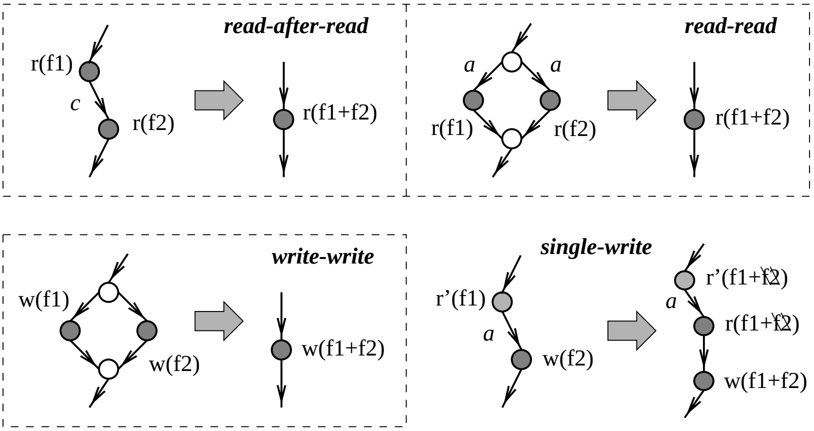


Figure 4.6. Relevant dependency cases

The notation used in the table is as follows:

- ‘*r(f1)*’ - read field ‘*f1*’, ‘*w(f2)*’ - write field ‘*f2*’;
- ‘*r(f1)*; *r(f2)*;’ - two sequential reads, ‘*r(f1)*, *r(f2)*;’ - two parallel reads;
- ‘*r(f1+f2)*;’ - reading two packed fields;
- ‘*r(...+~~f2~~)*;’ - a field is read an discarded (unused data);
- ‘*r(f1)*; / *r(f2)*;’ - mutually exclusive read operations.

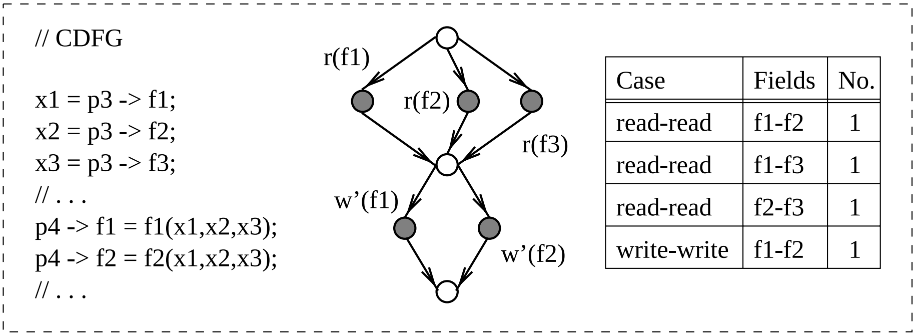
In Table 4.1, the number of accesses needed is presented together with the type. The column ‘change’ shows the change in the number of accesses per case after pre-packing.

A closer analysis of the eight possible cases shows that only three relevant cases are needed. This is especially true when the main target is the total reduction of the number of memory accesses. The reasons are:

- the cases “write-after-read”, “read-or-read” and “read-write” do not affect the total number of accesses and can be ignored;
- the cases “read-or-write” and “write-or-write” can be interpreted as independent writes (“single-write”), though these two cases are relevant when also the critical-path estimation is desired.

The remaining cases are as follows (see also Figure 4.6):

- *read-after-read* (speculative read) - reading field ‘*f2*’ (*r(f2)*) is control dependent on the reading of field ‘*f1*’ (*r(f1)*) and they can be merged (*r(f1+f2)*);
- *read-read* - two parallel reads (*r(f1)* and *r(f2)*) which use the same pointer (base address) can be merged (*r(f1+f2)*); and



```
// CDFG

x1 = p3 -> f1;
x2 = p3 -> f2;
x3 = p3 -> f3;
// . . .
p4 -> f1 = f1(x1,x2,x3);
p4 -> f2 = f2(x1,x2,x3);
// . . .
```

Case	Fields	No.
read-read	f1-f2	1
read-read	f1-f3	1
read-read	f2-f3	1
write-write	f1-f2	1

Figure 4.7. Collecting dependency cases

- *write-write* - two parallel writes (*w(f1)* and *w(f2)*) which use the same pointer (base address) can be merged (*w(f1+f2)*).

The two first cases are treated as a single one (*read-read*) since they affect the number of accesses exactly in the same manner. The case *write-write* is kept separately since write operations must be treated differently to avoid data corruption. A special case- *single-write* - is added to illustrate this. ‘*r'(f1)*’ and ‘*w(f2)*’ are used to denote that different base addresses are used for reading and writing. The cases to be collected are referred later as *read-read* and *write-write*.

To illustrate how the dependency cases are collected an example consisting of two independent pieces of the simplified CDFG, together with the corresponding C-code, is shown in Figure 4.7. In both cases, the data structure consists of three fields - ‘*f1*’, ‘*f2*’ and ‘*f3*’. ‘*r(f1)*’ and ‘*w(f1)*’ mark reading and writing the field ‘*f1*’, etc. Column “No.” in both tables shows the total number of corresponding dependency cases in CDFG. The same example is later used to illustrate the compatibility graph building and cluster grouping phases.

The long dependency chains and cases “read-or-write” and “write-or-write”, not analyzed in the scope of the proposed approach, are subjects of future research.

4.6. Compatibility graph construction

The compatibility graph of basic-groups (structure fields), *G=(N,E)*, is built based on the dependency cases. The nodes of the graph are later grouped hierarchically into clusters using a clustering method commonly referred as “hierarchical clustering” [GDW93]. The method is briefly described in the next section. The objective of the compatibility graph is to evaluate the closeness of the pairs of basic-groups - the “closer” the BG-s the more beneficial is their packing.

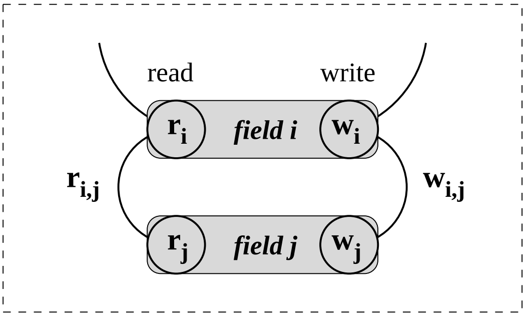


Figure 4.8. Compatibility graph definitions

Since the read and write operations behave slightly differently when merging two of them - writes may require additional fetching to avoid data corruption - they must be treated differently. Therefore every node ($n_i \in N$), each node corresponding to a BG, has two weights associated with it - one to represent writings (w_i) and another to represent readings (r_i). The value of a weight is equal to the number of corresponding accesses.

The sum of weights of all nodes gives essentially the total number of memory accesses - the main cost to be minimized. Using bit-widths together with the number of accesses allows finer tuning when also the uniformity of widths of memory words is part of the cost function. The number of accesses can be found based on the static analysis of the CDFG, though some assumptions should be made about execution probabilities of control branches. Another possibility is to use profiling information - every access operation should also carry data about how many times it was executed, or basic-groups should have data about the frequency of read and write accesses.

Any relevant pair of memory accesses forms a weighted edge in the graph. The edges are marked as $<r_i,r_j> \in E$ or $<w_i,w_j> \in E$ depending on the case of the access - “read-read” or “write-write”. The weight - $r_{i,j}$ or $w_{i,j}$ correspondingly - is equal to the number of corresponding dependency cases. The cases may create edges only between corresponding accesses:

- the “read-read” case creates an edge only between two read accesses (“read-side” of a node); and
- the “write-write” case creates an edge only between two write accesses (“write-side” of a node).

It should be noted that $r_i \geq r_{i,j}$ and $w_i \geq w_{i,j}$ because the number of cases related to two accesses can never be larger than the smallest of these two accesses.

Case	Fields	No.
read-read	f1-f2	1
read-read	f1-f3	1
read-read	f2-f3	1
write-write	f1-f2	1

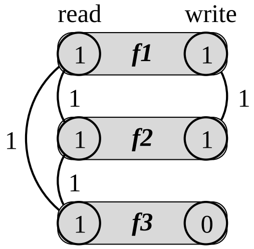


Figure 4.9. Compatibility graph example

The compatibility graphs corresponding to the example in Figure 4.7 are shown in Figure 4.9. For clarity, the nodes are placed in such a way that their “read-sides” are directed to the left. Every node is marked with its type - read/write and the field number - and weight, and every edge is marked with its weight.

These edges can be merged into hyper-edges, in principle, since usage of hyper-edges is more generic. This is left for future research because:

- the cases are collected pair-wise according to the dependencies; and
- in the clustering phase, the nodes are merged also pair-wise, and the usage of hyper-edges would result in modifying edges instead of simply removing them.

4.7. Clustering

The hierarchical clustering, also known as “hierarchical partitioning”, algorithm considers a set of objects and groups them according to some measure of closeness. The two closest objects are clustered first and are considered to be a single object for future clustering. The clustering process continues by grouping two individual objects, or an object or cluster with another cluster on each iteration. The algorithm stops when a single cluster is generated and a hierarchical cluster tree has been formed. Thus, any cut-line through the tree indicates set of sub-trees or clusters to be cut from the tree. [GDW93]

Hierarchical clustering has been used by many researchers for clustering in various VLSI design phases [KuCh97], [LaTh91], [McKo90].

As an example of the hierarchical clustering, let us consider a weighted graph depicted in Figure 4.10 (Step 1). The closeness of two nodes is defined as the weight between these two nodes -

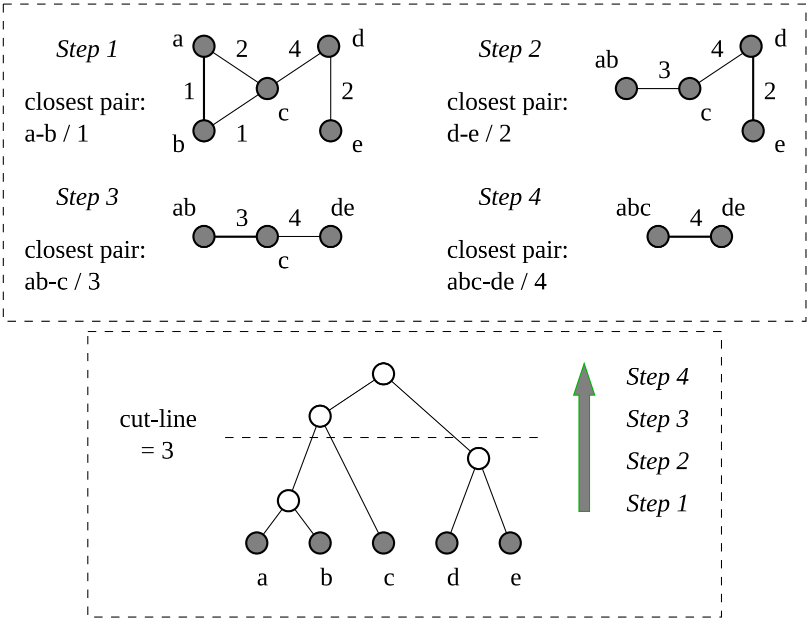


Figure 4.10. Hierarchical clustering

the smaller the weight the "closer" are the nodes. Merging two nodes into a cluster means effectively that the edge between these two nodes is removed. The weights of edges between these two nodes and the other nodes are reassigned as sums of the weights of the previous edges. The whole clustering process is shown step-by-step in the same figure together with the resulting cluster tree.

The cost function and the clustering algorithm have been slightly modified for the memory field pre-packing. The cost of the graph is equal to the sum of weights of nodes - the total number of accesses. The "closest pair" is defined as the pair of accesses which gives the best improvement of the cost function, i.e. which gives the greatest reduce in the number of accesses. There exist different ways to find the closest pair. One of the ways, of course, is to merge nodes pair by pair, calculate the new cost and select the best improvement. Since merging modifies only weights of the nodes to be merged, it is sufficient to find the change of the node weights only. A set of heuristic re-evaluation rules of weights, described below, has been developed to avoid rebuilding the CDFG and compatibility graph after every clustering iteration. The weights of nodes and edges are re-evaluated as follows:

- let the nodes to be merged are n_i and n_j, and the new node is n_k;
- read-weight - $r_k = max(r_i,r_j) + (w_i-w_{i,j}) + (w_j-w_{i,j})$ - takes into account that all fields are loaded anyway, and the number of extra reads needed before writings;
- read-edge-weight - $r_{k,n} = max(r_{i,n},r_{j,n})$ - the new number of cases "read-read";

- write-weight - $w_k = max(w_i, w_j)$ - takes into account that all fields are stored anyway;
- write-edge-weight - $w_{k,n} = max(w_{i,n}, w_{j,n})$ - the new number of cases “write-write”.

The re-evaluation rules are based on the last two cases in Table 4.1 - “single-read” and “single-write” - and take into account how an extra field access affects the total number of accesses.

The clustering algorithm can be outlined as follows:

- two nodes, the merger of which give the best improvement of the cost function, are selected;
- the selected nodes are replaced with a single one, and the edges and weights are modified according to the re-evaluation rules above;
- the previous steps are repeated until all nodes have been merged into a single cluster (or only a single node remains), or the cost has reached a pre-defined value - “cut-line”.

Figure 4.11. demonstrates the cluster tree building of the example graphs from Figure 4.7 and Figure 4.9. The packing of two fields *f1* and *f2* is reflected in the labels of read and write operations of the simplified CDFG - *w(f1+f2)* and *r(f1+f2)* - and in the labels of the nodes of the compatibility graph - also *w(f1+f2)* and *r(f1+f2)*. The other field packings are reflected in a similar manner.

In Figure 4.12., a counter example is presented where the heuristics re-evaluation rules give one read too many. The upper example has two independent access groups which use different pointers (base addresses), and the lower example has three access groups. However, both CDFGs give exactly the same compatibility graph the upper example has one read less after merging data fields. This is caused by the fact that the compatibility graph does not take into account that there can exist complex independent access groups. The model behind the compatibility graph treats the accesses pairwise, in principle. The counter example was discovered when experimenting with real design examples and the upper example, presented in Figure 4.12., is a generalization.

There exist different ways to overcome the drawback. The first solution is, of course, to rebuild the CDFG at every clustering step to get the correct number of accesses. This is undesirable since it unnecessarily slows the clustering phase, and the heuristic re-evaluation rules were developed to avoid rebuilding of the CDFG at every step. Another solution is to expand the compatibility graph to represent independent access groups. Such a graph would have as many sub-graphs as there are different base-addresses - one sub-graph per base-address (pointer). The only condition is that the clustering is applied to all sub-graphs at the same time. The expansion allows not only more precise estimation of the access count, but also allows to analyze accesses to a shared memory from different sub-modules of a design. The drawback is, of course, the increased complexity of the clustering phase. The modified clustering of the counter example from Figure 4.12. is illustrated in Figure 4.13.

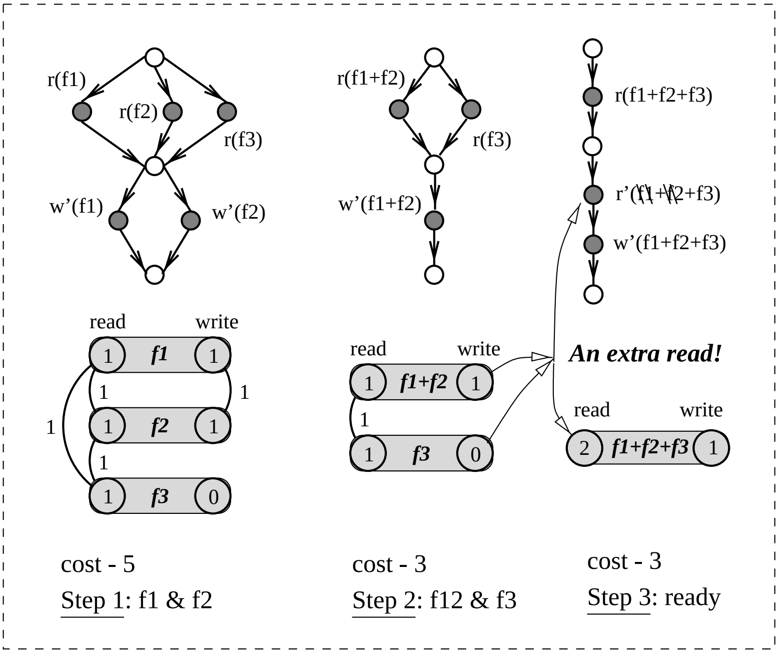


Figure 4.11. Clustering example

Additionally the total width of fields in bits can be used as an extra cost or as a constraint for both compatibility graph types. The cost function can be modified to evaluate also the uniformity of the field widths. Field splitting, another possibility to make the word width uniform, can be performed using the same clustering principles described above. A necessary modification of the algorithm is to allow splitting of nodes - the weights of the nodes and edges remain the same but the width is halved.

4.8. Results of experiments

Four sub-tasks from real-life ATM cell processing applications ([HSE95], [SWC97], [SYM98] and [WCD96]) have been used as test examples for pre-packing. The functionality of all examples is characterized by intensive transfer of dynamically allocated data structures. Exploration results after memory allocation and assignment phase are shown in Table 4.2. “Cycle budget” is the number of clock steps on the critical path. Relative area and relative power consumption are measured against the solutions without pre-packing (if applicable). The basic-groups were counted (“# of BG-s”) before storage bandwidth optimization, i.e. after pre-packing. The num-

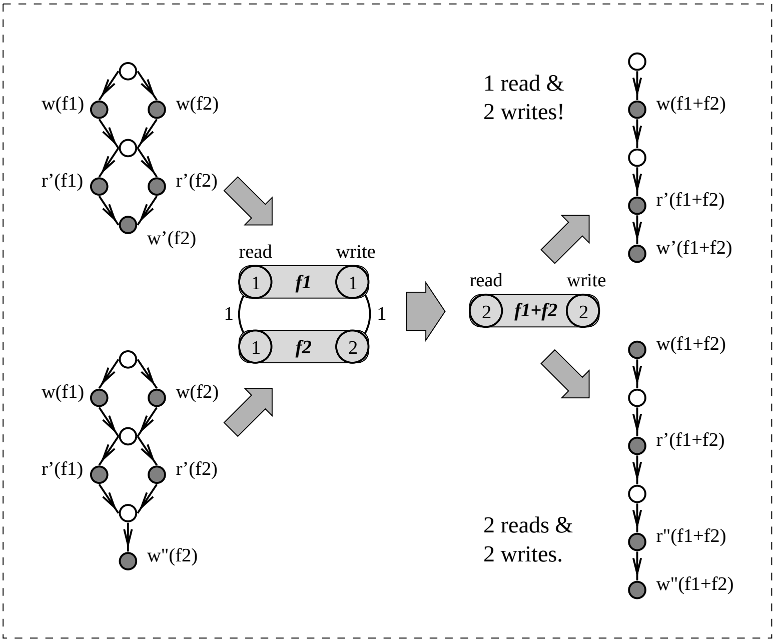


Figure 4.12. Counter example for the simple clustering heuristic

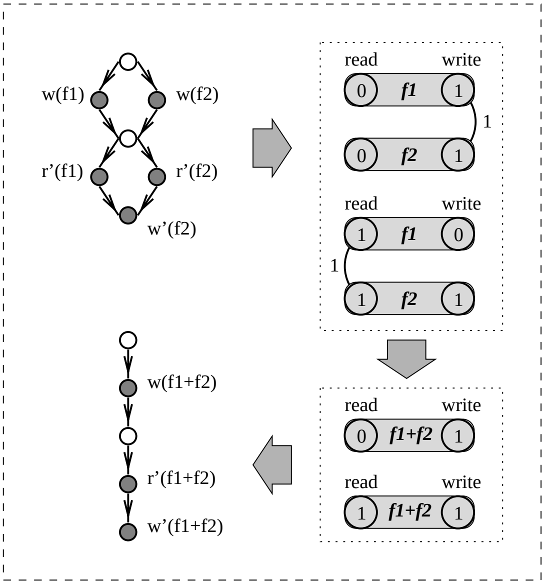


Figure 4.13. A solution for the counter example

ber of memories shows the number of physical memories after memory allocation phase. The used cost function estimates relative area and power consumption of memories, based on realistic memory architectures [WCD96], [LiHe98], [ShCh99].

Table 4.2. Results after memory allocation

Design		Number of accesses	Number of BGs	Cycle budget	Number of memories	Relat. area	Relat. power
#1	without pre-packing	18	19	12	4	1.0	1.0
	with pre-packing	6	6	4	4	0.798	0.397
#2	without pre-packing	85	44	50	8	1.0	1.0
	with pre-packing #1	41	15	30	8	0.899	0.582
	with pre-packing #2	39	14	30	8	0.881	0.572
	with pre-packing #3	38	17	30	8	0.907	0.552
#3	without pre-packing	234	49	-	-	-	-
	with pre-packing #1	149	22	29	8	1.0	1.0
	with pre-packing #2	156	19	34	7	1.011	0.787
#4	without pre-packing	219	23	-	-	-	-
	intuitive packing	76	16	-	10	1.0	1.0
	with pre-packing #1	36	8	-	6	0.970	0.592
	with pre-packing #2	35	7	-	6	0.970	0.586
	with pre-packing #3	35	6	-	5	0.958	0.567

The cases of design #3, Segment Protocol Processor (SPP), show how exploration at higher levels can improve results. The first alternative by designer was *pre-packing #1*. However, by exploring designer could get another combination of fields that although increasing the critical path and number of accesses, it finally lead to an improvement in overall power of more than 20% (*pre-packing #2*).

Design #4 illustrates how the pre-packing methodology affects storage characteristics of Operation and Maintenance (OAM) handler of ATM switch. High-level synthesis results of the driver are described in details in [HSE95]. The cases presented in the Table 4.2. are:

- *without pre-packing* - initial simulatable specification in VHDL.
- *intuitive packing* - synthesizable by high-level synthesis tools, partitioned manually into handable processes. Some manual optimizations of accesses, based on intuitive pre-packing, were performed during rewriting.
- *pre-packing #1* and *pre-packing #2* - pre-packing methodology applied for the synthesizable code. The cases differ in packing illustrated in Figure 7.9, while keeping the total number of

memories the same.

- *pre-packing #3* - the cross sub-module dependency analysis allowed additional merging, thus reducing the memory waste (the number of accesses remained the same).

Exploration of different packings of the design #4 is discussed in more details in Chapter 7.2.

For smaller examples (e.g., drivers #1 and #2), the proposed methodology gave a very similar packing result compared to the ones obtained when using an ad-hoc manual approach. For larger ones (e.g., drivers #3 and #4) it clearly outperformed the ad-hoc approach. This substantiates the validity of assumptions and the effectiveness of the proposed technique.

Some points of interests, suggested by the clustering, were explored for further optimization, but the final area/power figures did not change much. The solutions differed only how one or another, not so intensively accessed, data fields was packed. Very small differences in area and power consumption point out the need for accurate estimates for finer trade-offs [VMB99].

4.9. Conclusion

The memory access bandwidth is one of the main design bottlenecks in data-transfer dominated applications such as protocol processing and multimedia. In this chapter, a systematic technique is proposed which reduces memory bandwidth by rearranging the layout of the dynamically allocated data records stored in memory. At the same time memory bit-waste is kept as low as possible. The technique exploits parallelism in the data-transfers by analysis of the dependencies between memory accesses.

The proposed technique allows system designers to explore different memory data-layout alternatives at the system level when the application is still being defined. It aims mainly to minimize the total number of data-transfers by merging memory accesses, and the total storage is minimized by reducing bit-width wastage.

Four subsets from real-life ATM cell processing applications have been used as test drivers for the methodology. The memory size was reduced by up to 20% and the power consumption by 40% to 60%. The results are very similar to the ones developed manually by experienced designers for smaller designs, but significantly better for large ones. The automation of these techniques will also further reduce the design time.

5. Segment-Based Scheduling

In this chapter, a control flow based scheduler is described. The scheduler splits control flow into segments while traversing it thus avoiding path explosion typical for path based schedulers. Very efficient schedules of loops can be achieved because the scheduler does not break loops but traverses them in the first order.

5.1. Introduction

Scheduling is one of the key steps in High Level Synthesis (HLS) and a significant portion of the HLS research, which spans more than a decade now, has been devoted to the problem of scheduling. As the HLS research moved into 90’s, it was realized that the scheduling techniques developed earlier were not in a position to take into account the requirements of the control dominated circuits encountered in areas such as telecommunication. These requirements fall into two major areas:

- **Control constructs:** The control dominated circuits typically have a complex control flow structure, involving nesting of loops and conditionals. Therefore, in order to get a high quality design, the entire behavior inclusive of the control constructs needs to be considered, rather than just the straight-line pieces of code corresponding to the basic blocks. In particular, loops deserve a special attention in order to optimize performance. The schedulers developed for data path dominated circuits primarily focus on computations within a basic block. In some cases, the extensions of the basic technique to consider loops or conditionals to a limited extent have also been reported but these are not designed to handle arbitrary nesting of control constructs.
- **Operation chaining:** In control dominated circuits, there is a large number of condition checking, logical and bit manipulation operations. The propagation delays associated with these operations are much smaller as compared to arithmetic operations. Therefore, there is a large opportunity for scheduling multiple operations (data as well as control) chained together in a control step and ignoring this would lead to poor schedules. The schedulers developed for data path dominated circuits usually consider operator delays as multiple of clock periods. Chaining of data operations is considered in some cases as an extension of the basic scheme and not as the primary consideration. In any case, chaining of control operations with data operations is rarely considered.

Path Based Scheduler [CaBe90], [Cam91] was the first attempt to address these requirements. It works with Control Flow Graph (CFG) instead of Data Flow Graph (DFG), builds all possible control paths, schedules each of them separately following the AFAP strategy, and finally merges all different paths into one Finite State Machine (FSM). The main drawback of the approach is the need to evaluate all possible paths, which number can be enormous. Different improvements have been made to reduce the number of paths [BRN97], [HLH93], and to improve loop scheduling [BDB94], [ROJ94], [YeWo96].

Several schedulers have been developed for High Level Synthesis of control dominated systems but there is no scheduler which solves the problem comprehensively and efficiently, considering all the relevant issues - operation chaining, control constructs handling, to name some of them. In this chapter, a scheduler, which essentially meets these requirements, is presented. Two major components of the scheduler - procedures for traversal of control flow/data flow, and for analyzing where to locate the state boundaries - avoid construction of all control paths (or their tree representation) before scheduling. This is achieved by dividing control graph into segments during the graph traversal. Segmentation drastically reduces the number of possible paths thus speeding up the whole scheduling process because one segment is analyzed at a time only.

In the next section, related work about scheduling of control dominated algorithms is described. In the third, section the used control-flow representation is described. The fourth and the fifth sections focus onto two major problems of the scheduling - how multiple paths are traversed, and how the loops are handled. In the sixth section the state marking rules are discussed, and in the seventh results of comparison of different scheduling approaches are presented.

5.2. Related work

Path Based Scheduling (PBS) was introduced by [CaBe90], [Cam91] as a mechanism to meet these scheduling requirements. In order to meet the first requirement, the PBS works with the control flow representation of behavior rather than the data flow representation used by earlier schedulers. Secondly, the problem of scheduling is formulated as a problem of partitioning each of the control flow paths into a sequence of control steps, which brings focus on operation chaining and allows easy exploration of the various chaining possibilities. PBS is able to achieve an as-fast-as-possible schedule for all the paths at a global level, possibly at the cost of duplication of operations in different control steps. However, as it does not take into account data flow or data dependency information, the operations can not be reordered or parallelized. Thus, the solution is optimal only for the given order of operations. Another drawback of this approach is that a vast amount of time and space is required to enumerate all possible paths.

The problem of path explosion in PBS is somewhat reduced in tree-based scheduling [HLH93] which also produces an as-fast-as-possible schedule for all paths. The rooted tree used here to represent the collection of paths is more compact as compared to considering all the paths individually, but can still be exponentially large. The tree based scheduler considers data dependencies and thus it is not constrained by the initial order of the operations. It also considers resource constraints but no constraints involving operator delays and i/o event ordering are considered.

A different approach is presented in [WBL94], which examines control flow as well as data flow and formulates the scheduling constraints as linear (in)equalities. This results in an ILP problem with a special form, which lends itself to a polynomial time solution. However, control flow paths (and data flow chains in this case) need to be traced during ILP formulation. One strong feature of this approach is that it permits moving of operations across the basic blocks.

A major enhancement of PBS is reported in [BRN97]. It takes care of operation ordering by examining data dependencies and applying a data flow based scheduling technique (List Scheduling, in particular) locally. The number of paths to be considered is reduced by placing cuts in the control flow graph. A cut introduces a control step boundary. The location of a cut is decided on the basis of the extent to which it can reduce the number of paths and satisfy the scheduling constraints. Cuts are introduced repeatedly until the number of paths reduces to the desirable level.

In all these approaches targeted for scheduling of control dominated circuits, one important aspect is still ignored - loops in the control flow. In all these, the loops are cut open in the beginning by removing the feedback edges in the control flow graphs and closed finally after scheduling by replacing the removed edges. Thus, control flow paths that go around the loops are ignored. It was demonstrated in [BDB94] with some examples that a proper treatment of loops is essential because scheduling of loops is usually critical in determining the overall performance. They presented a measure of overall performance of a behavioral specification based on branching probabilities and described an algorithm called loop-directed scheduling (LDS) which traverses all the control flow paths, including those which go around the loops. The LDS algorithm, however, does not consider input/output and timing constraints, does not allow chaining of data operations and relies on the given order of the operations in the behavioral description.

A similar algorithm called dynamic loop scheduling (DLS) has been reported in [ROJ94], [RaJe95], [RaJe95b]. Like the LDS algorithm, the DLS algorithm traverses all paths including loops and it also does not allow chaining of data operations and relies on the given order of the operations in the behavioral description.

A method which focuses on efficient scheduling of loops in the control flow has been presented in [YeWo96]. The behavioral description, which is input to this scheduler, is in the form of an abstract state machine, which may have arbitrary flow of control. It can accept a variety of timing constraints going across many basic blocks, including constraints involving multiple iterations of a loop. The scheduling algorithm is based on cylindrical layout compaction and can find an optimal schedule of a single loop in polynomial time. Scheduling for arbitrary control flow graphs is shown to be NP-Complete and a heuristic solution is provided. Though the scheduler is targeted for control dominated circuits, it does not consider chaining of operations.

Clearly, there are several schedulers which have been developed for High Level Synthesis of control dominated circuits. However, there is no scheduler which solves the problem comprehensively and efficiently, considering all the relevant issues. Namely, performing operation chaining, handling control constructs including loops globally, exploit data dependence (rather independence), consider variety of constraints (input/output constraints, timing constraints, resource constraints and clock period constraints) and optimization criteria (performance, area, power etc.). In this chapter, a scheduler is presented, which essentially meets these requirements.

The segment based scheduler has two major components: an efficient procedure for traversal of control flow/data flow and a comprehensive procedure for analyzing where to locate the state boundaries taking into account a variety of requirements and constraints. It avoids construction of all control paths (or their tree representation) before scheduling. Instead, during the scheduling processes the control graph (or flow chart) is traversed and divided into segments as the scheduling proceeds. Each segment represents a collection of paths from one state into others. Segmentation drastically reduces the number of possible paths thus speeding up the whole scheduling process. This reduction is achieved by analyzing one segment at a time only in contrast to the path-based and tree-based scheduling which analyses all possible paths. Segment in the approach is a path in the CFG from one state into another. Two different scheduling strategies can be outlined - state and transition oriented. In the state oriented scheduling operations shared by different segments of the CFG are treated in the same manner and consequential scheduled into the same control step. In the transition oriented scheduling, these operations are treated independently, thus allowing to schedule them into different control steps, at the cost of duplication of these operations.

Various data flow based scheduling techniques and transformations can be used to improve control flow centric schedulers. The best candidates are techniques, which allow to move operations out of basic blocks, thus allowing speculative execution, for instance. Another benefit of allowing operations to be moved is creating incremental improvements of the overall schedule. [FRJ94], [RaBr96], [BRN97], [LRJ98]

5.3. Control flow/data flow representation and overview of scheduling process

Primarily an explicit control flow information is required so that global control flow can be easily analyzed. The usual Control and Data Flow Graph (CDFG) [GDW93] used by most HLS tools is not suitable as it primarily captures data dependencies and does not lend itself for a convenient control flow analysis. Data dependency information is not needed in order to prevent the operations to be artificially constrained by the order inherent in the control flow. In [BRN97] the data dependency information is used locally, at the basic block level, to optimally reorder the operations. Whereas in [YeWo96] and [WBL94], the data dependency information is used at a global level and the operations can be reordered and can even move across the basic blocks. In [WBL94] since loops are opened before scheduling, the data dependencies are also acyclic and are handled without difficulty. In [YeWo96], on the other hand, wrapping numbers are used to distinguish between the dependencies that go around the loops from those which don’t. A two- tier approach, similar to that used in compilers (e.g. [ASU86]), has been adopted. This allows the local data dependencies to be used to reorder the operations within basic blocks, and global data flow analysis can be used to move the operations across the basic blocks if it turns out to be beneficial.

A flow chart like internal representation called IRSYD (see chapter 3) is used to represent the control flow. For segment based scheduling, a synthesizable subset is needed only. The needed subset describes control flow and data dependencies of a process. The Control Flow Graph (CFG) is a directed graph $G=(N,E)$. The nodes $n \in N$ represent control blocks to be scheduled and the edges give the precedence relation; i.e. $<n_i,n_j> \in E$ iff n_i is an immediate predecessor of n_j, and explicitly represent the control flow. The following four control flow node types of IRSYD are used:

- *entry* and *exit* nodes represent the start and end points of a control flow (process);
- *operation* nodes encapsulate all the computational activity of the CFG (associated with a data flow graph);
- *condition* nodes describe decision making and branching on a boolean variable containing the result of a condition evaluation.

Each node, except *exit* and *condition,* can have only one immediate successor. Whereas *exit* has no successors, *condition* has exactly two successors (for the present, but can be generalized without any difficulty) - one representing branch for *true* and another for *false*. A sub-graph is created for each VHDL function and/or VHDL process with an *entry* node and an *exit* node. The hierarchy, implied by functions, is flattened or sub-graphs are moved out as sub-controllers as directed by user. Using data flow analysis, parallel independent threads of control flow can be extracted and moved out as sub-controllers. This is needed because the underlying Mealy FSM has exactly one active thread. [EHK96], [EKS96]

Each edge of a CFG can possibly have one or both of the following types of special events, associated with the system clock, called markings:

- *wait* mark describes waiting for the active clock flank in the given HDL description (*wait* statement in VHDL for instance);
- *state* mark represents a state of the Mealy FSM corresponding to the CFG, as decided by the scheduling process.

The *wait* marks are present in the CFG before scheduling. These are derived from the behavioral description. The *state* marks are put by the scheduler. As it is possible to move some of the operations across the wait statements, without changing the semantics, it is possible to have the wait statements in the scheduled behavior at positions different from those in the unscheduled behavior. This effectively means that the edges carrying the *wait* marks are potential candidates for state marking but it is not necessary that these edges should be state marked. The essence of scheduling is putting state marks on the edges of CFG, as these marks decide the state boundary. The rules for deciding where to put these marks are based on compliance with the language se-

```
-- mem - data to be sorted,
--       size max_addr+1 words

for j in 1 to max_addr loop         -- 1,2,12
  for i in max_addr downto j loop   -- 3,4,11
    if mem(i-1) > mem(i) then       -- 5,6,7
      tmp := mem(i-1);              -- 5,6
      mem(i-1) := mem(i);           -- 7,5,9
      mem(i) := tmp;                -- 10
    end if;
  end loop;
end loop;
```

Figure 5.1. VHDL code of Bubble sort

mantics, design constraints and optimization criterion. These rules are discussed in section 5.6.

A state transition from state s_i to s_j corresponds to the set of directed path segments between the pair of edges carrying the *state* marks for s_i and s_j. As the states are marked on the edges and actions/ conditions are associated with nodes, the action for the state transition are the actions associated with the nodes in these path segments. As the path segments may include conditions, the actions may be conditional. Thus, a CFG represents a Mealy machine and allows explicit description of complex decisions and conditional actions to be executed within a state or a clock cycle.

As an example, an algorithm implementing the bubble-sort is used (see Figure 5.1.). The data to be sorted has been already stored into memory **mem**, and saving/reading of that data is not shown. The code has been optimized to reduce the number of memory accesses, and is shown on Figure 5.2. together with corresponding part of CFG. Operations in the code are marked with numbers and corresponding numbers are shown in the nodes of CFG.

5.4. Traversing CFG

Paths in CFG are traversed by a recursive, depth-first search procedure called `FollowPath`. The starting point for the traversal is the entry node of the overall process, which forms the initial state. As paths are traversed further states are marked and added to the **state_list**. From each state, traversal is continued further until no more states are marked. The overall procedure is shown in Figure 5.3.

The procedure `FollowPath` is called recursively until another state mark is encountered or a back edge of a loop is encountered. The path traversed (**edge_list**) is then passed on to the pro-

```
j := 1;                     -- 1
while j <= max_addr loop    -- 2
  i := max_addr;            -- 3
  while i >= j loop         -- 4
    ix := i-1;              -- 5
    w1 := mem(ix);          -- 6
    w2 := mem(i);           -- 7
    if w1 > w2 then         -- 8
      mem(ix) := w2;        -- 9
      mem(i) := w1;         -- 10
    end if;
  i := ix;                  -- 11
  end loop;
  j := j+1;                 -- 12
end loop;
```

All identifiers are variables except memory **mem**.
Complex operations divided into elementary ones:
6 - reading from memory - **6′** and **6″** - sending address to **mem** and reading data;
7 - reading from memory - **7′** and **7″** - sending address to **mem** and reading data.

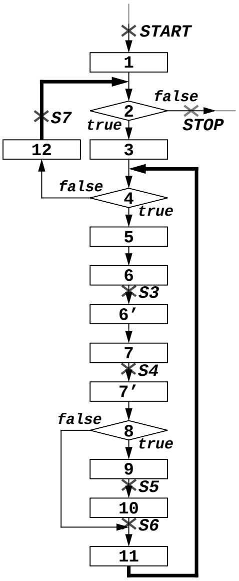


Figure 5.2. Optimized VHDL code and corresponding CFG

```
procedure MarkStates begin
  state_list := {process_entry_edge};
  new_state := nil;
  while ∃ <n1,n2> ∈ state_list, which has not been processed
  loop begin
    FollowPath ( <n1,n2> , ∅ );
  end loop;
end procedure;
```

Figure 5.3. Procedure *MarkStates*

cedure `AnalyzePath` which returns the edge on which the state mark is put, as shown in the Figure 5.4.

One problem with `FollowPath`, as described above, is that the work done in tracing the path beyond the point where a state mark is later inserted by `AnalyzePath`, is actually wasted. That part of the path eventually gets traced again in a later depth-first traversal session, starting from the new state mark. More seriously, if there are branches in that part, it will get partially traced several times and passed on to `AnalyzePath` while trying to complete the current session of

```
procedure FollowPath ( $<n_1,n_2>$ , edge_list ) begin
  if  $<n_1,n_2>$ is not already marked and
      $n_2$ does not lie on edge_list  then begin
    edge_list := edge_list $\cup$ { $<n_1,n_2>$ };
    for $\forall$ $<n_2,n_x> \in E$ loop begin
      FollowPath ( $<n_1,n_x>$ , edge_list );
    end loop;
  else begin
    new_state := AnalyzePath ( edge_list );
    state_list := state_list $\cup$ new_state;
  end if;
end procedure;
```

Figure 5.4. Procedure *FollowPath*

```
        (* . . . *)
        for $\forall$ $<n_2,n_x> \in E$ loop begin
          FollowPath ( $<n_1,n_x>$ , edge_list );
          if   new_state $\in$ edge_list   then   return;
        end loop;
        (* . . . *)
```

Figure 5.5. Modification in *FollowPath*

depth-first traversal, without leading to new state marks. The latter problem can be, however, easily remedied by checking whether the node corresponding to the recently marked state is in `edge_list` and terminating path tracing and analysis in case it is so. The modified for - loop of `FollowPath` procedure now is given in Figure 5.5.

A better approach is to interleave `AnalyzePath` and `FollowPath` more closely. That is, for every step moved by `FollowPath`, `AnalyzePath` is called to check if a state mark should be inserted. If `AnalyzePath` decides not to put a mark, only then the path is traversed further by calling `FollowPath`. This avoids path tracing beyond the state mark. The resulting procedure, called `FollowPathState`, is shown in Figure 5.6.

Now consider two paths, which partly overlap. In the procedure, developed above, the overlapping part will have a common schedule. However, if it is desired to have optimal schedule for each path independently (e.g. As Fast As Possible for each path), the condition which prevents path traversal beyond a state mark can be omitted. Now two consecutive state marks on a path will not necessarily correspond to a state transition. Therefore, there is a need to explicitly record the state transitions as the state marks get decided. The procedure, which achieves this, called `FollowPathTransition` is shown in Figure 5.7., along with the main procedure containing some additional initialization.

```
procedure FollowPathState ( $<n_1,n_2>$ , edge_list ) begin
  if $<n_1,n_2> \notin$ state_list and $n_2$ does not lie on edge_list
  then begin
    edge_list := edge_list $\cup$ { $<n_1,n_2>$ };
    new_state := AnalyzePath ( edge_list );
    if new_state = nil   then begin
      for $\forall$ $<n_2,n_x> \in E$ loop begin
        FollowPathState ( $<n_1,n_x>$ , edge_list );
      end loop;
    else begin;
      state_list := state_list $\cup$ {new_state};
    end if;
  end if;
end procedure;
```

Figure 5.6. Procedure *FollowPathState* (state oriented)

```
procedure MarkStates begin
  state_list := {process_entry_edge};
  trans_list := $\emptyset$ ;
  new_state := nil;
  while $\exists$ $<n_1,n_2> \in$ state_list, which has not been processed
  loop begin
    current_state := $<n_1,n_2>$;
    FollowPathTransition ( $<n_1,n_2>$ , $\emptyset$ );
  end loop;
end procedure;

procedure FollowPathTransition ( $<n_1,n_2>$ , edge_list ) begin
  if $n_2$ does not lie on edge_list   then begin
    edge_list := edge_list $\cup$ { $<n_1,n_2>$ };
    new_state := AnalyzePath ( edge_list );
    if new_state = nil  then begin
      for $\forall$ $<n_2,n_x> \in E$ loop begin
        FollowPathTransition ( $<n_1,n_x>$ , edge_list );
      end loop;
    else begin
      state_list := state_list $\cup$ {new_state};
      trans_list := trans_list $\cup$ {(current_state, new_state)};
    end if;
  end if;
end procedure;
```

Figure 5.7. Procedure *FollowPathTransition* (transition oriented)

The result of state marking for overlapping paths in procedure `FollowPathState` depends upon the order of traversal of the paths. Consider the situation shown in Figure 5.8.a) with overlap-

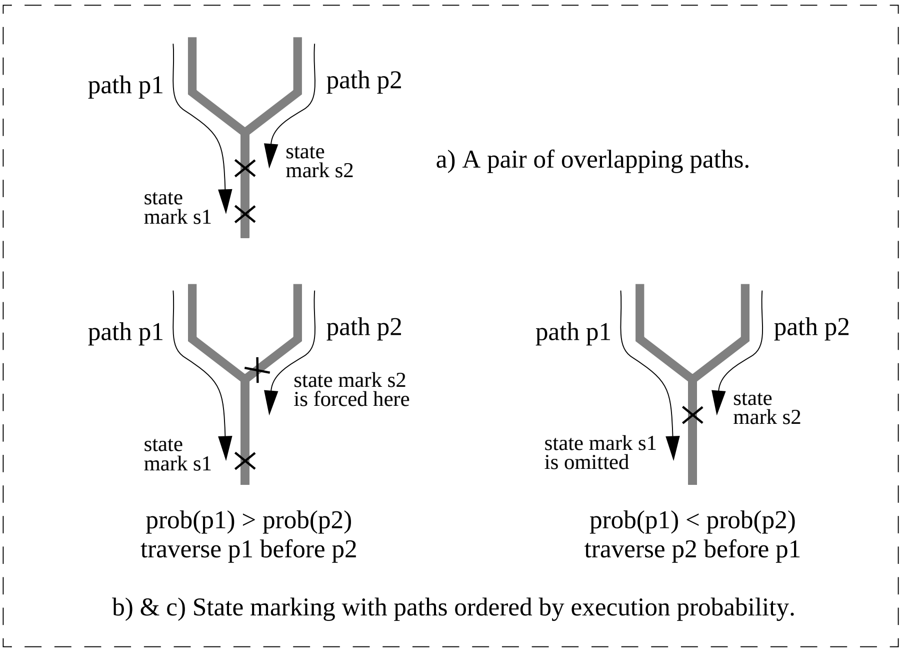


Figure 5.8. State marking of overlapping paths

ping paths p1 and p2. Suppose independent traversal of p1 and p2 leads to state marks s1 and s2 respectively in the overlapping portion as shown in the figure. Now, using procedure `FollowPathState`, if p1 is traversed before p2, both the marks are put and it may have an effect of slowing down the execution of both the paths because of an extra state mark on each path. On the other hand, if p2 is traversed before p1, s2 is the only state mark which results. Thus, execution of p2 is not effected but that of p1 may be slowed down as the state mark is appearing earlier than its best position. It is possible to confine this slowing effect to the path, which has a lower execution probability (an approach to find these probabilities is given in [BDB94]). This can be done by ordering the outgoing branches of a node in decreasing order of their branching probabilities and preventing state marks to be put on a previously traversed path. The resulting state marks for the example of Figure 5.8.a) are shown in Figure 5.8.b) and c).

Both the procedures presented above (`FollowPathState` and `FollowPathTransition`) are quite efficient as they avoid construction of all the paths from the entry node to the exit node. However, they still need to construct all the path segments from one state mark to the location of the next potential state mark. If the behavioral description contains a sequence of conditional assignments, which can potentially be scheduled in a single state, the number of the path segments will also tend to be fairly large. For example, in the graph of Figure 5.9., there are 9 path segments between node A and node B. Such local explosion of path segments can be easily handled by path collapsing described in [BRN97].

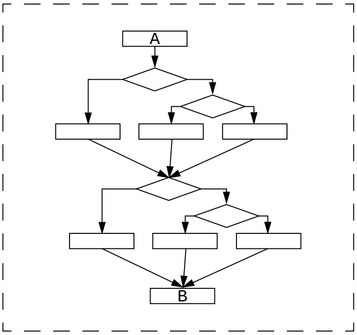
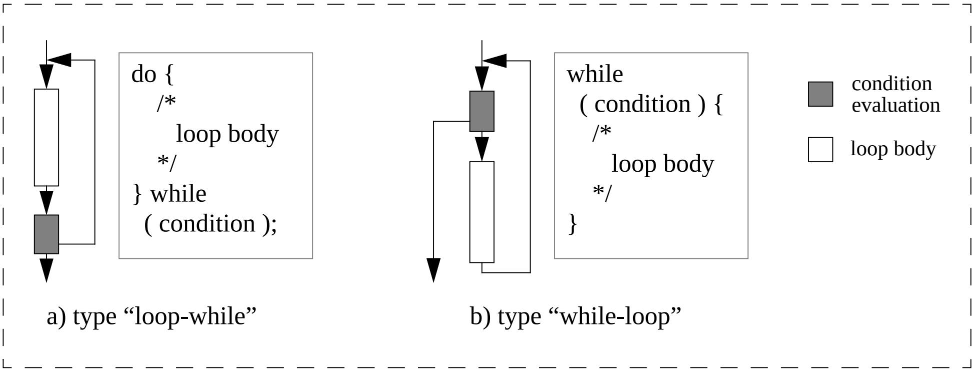


Figure 5.9. A part of a CFG with nine path segments



```
do {
   /*
     loop body
   */
} while
  ( condition );
```

```
while
  ( condition ) {
    /*
      loop body
    */
}
```

a) type “loop-while”

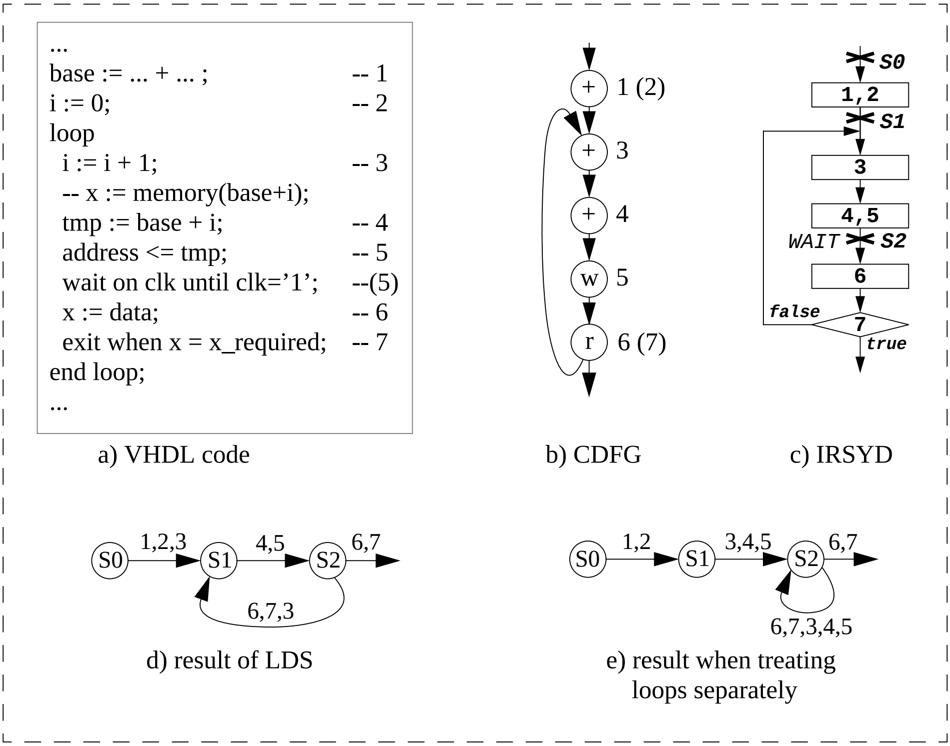
b) type “while-loop”

Figure 5.10. Loops generated from High-Level Languages

5.5. Scheduling loops

Control-flow in behavioral synthesis is usually extracted from description in High-Level Language where two basic types of loops can be found (see Figure 5.10.). On the first type of loops (“loop-while”) the strategy of breaking loops by removing feedback edge, used by most of the control oriented scheduling strategies ([CaBe90], [Cam91], [HLH93], etc.), works without destroying the control-flow. This simple strategy can not be applied onto the second type of loops (“while-loop”) without a need to modify the resulting graph because breaking the feedback edge leaves the loop body without control successors.

The scheduling strategy, LDS (Loop Directed Scheduling), proposed in [BDB94] allows to take into account also the feedback edges but this approach can still result in inferior schedules because of the way the loops are treated. Let us consider a simple search example shown in Figure 5.11. Here operations 1, 3 and 4 are of similar type; i.e. they can be bound onto the same func-



```
...
base := ... + ... ;                -- 1
i := 0;                            -- 2
loop
  i := i + 1;                      -- 3
  -- x := memory(base+i);
  tmp := base + i;                 -- 4
  address <= tmp;                  -- 5
  wait on clk until clk='1';       --(5)
  x := data;                       -- 6
  exit when x = x_required;        -- 7
end loop;
...
```

a) VHDL code

b) CDFG

c) IRSYD

d) result of LDS

e) result when treating loops separately

Figure 5.11. Loop scheduling of a search example

tional unit (adder for instance). Operations 5 and 6, from other side, are of type which enforces a clock flank (state mark) between these two operations - synchronization (wait statement), memory access, etc. Let us assume that two adders can be used and both are available when LDS algorithm arrives to the operation 1 (direction from the beginning of the control flow). Operations 1 and 3 will be scheduled into one and 4 into another transition because of the resource constraint. Although only one adder will be used when scheduling the following operations (for 4) operations 5 and 6 will be scheduled into different transitions because of the enforced clock flank - access to the external memory. The resulting state diagram is shown in Figure 5.11.d. where it can be seen that every loop iteration takes two clock cycles.

A different schedule where every iteration take exactly one clock cycle is shown on Figure 5.11.e. This improvement is achieved scheduling first the loop body - operations 3 and 4 could be scheduled into the same transition because both adders are available. Operations 5 and 6 must be still separated by a clock flank but they still can be scheduled into the same transition because the effectively belong to the different iterations of the loop. In practice the operation 3 can be scheduled both into transition from S0 to S1 and into transition from S1 to S2. The later solution results how entry and exit edges of loops are treated and is discussed below.

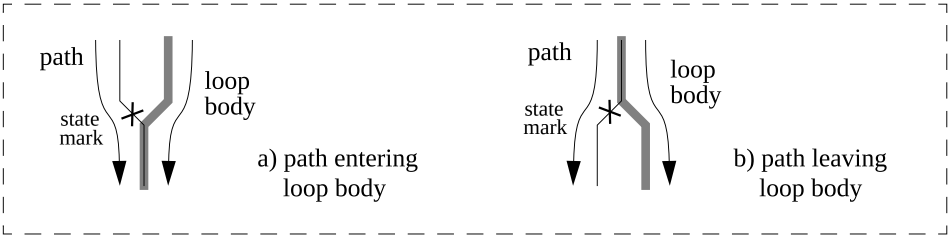


Figure 5.12. Path partly inside loop body

One of the main concerns when scheduling loops is where and/or how to break loops to get an acyclic graph which then can be scheduled using simpler algorithms. As it has been shown in related works, and can be seen from the example described above, it is not a trivial task to break loops to achieve efficient schedules. The approach described in this chapter is based on the following assumptions [BDB94]:

- loops are the most critical parts of an algorithm from the performance point of view and therefore they should be considered in the first order;
- constraint conflicts when scheduling loop entries and exist are preferably solved by creating extra states outside the loop because this affect less the overall performance.

The loops require a special treatment in scheduling as illustrated above. The procedures for CFG traversal described in the previous section do not treat loops specially, though they traverse the entire CFG including loops. In relation to a loop, a path segment constructed by these procedures may

- begin outside the loop and end inside its body, or
- begin inside the loop body and end outside the loop, or
- be totally out side the loop, or
- be totally inside the loop but exclude the back edge of the loop (i.e., contained within an iteration), or
- be totally inside the loop but include the back edge of the loop (i.e., go from one iteration to the next iteration).

When scheduling segments, which are partly inside a loop body, the following restrictions are applied (Figure 5.12.):

- state mark on a path entering the loop is forced outside the loop body - an extra state mark caused by the clock period or resource constraints outside the body reduces performance of the loop (see also Figure 5.11.);
- state mark on a path leaving the loop is also forced outside the loop body.

Forcing states outside loop body can be done because any constraint violation inside the body is solved when scheduling paths, which are totally inside the loop body.

In order to treat the loops separately, they need to be scheduled first. This means that the path segments of the last two types (which lie entirely within a loop body) should be constructed first. In `FollowPath` procedures, if some initial state mark for each loop is put, then traversal can be started from those marks and complete state marking of the loops by confining the traversal to the loop bodies. The question now arises, where the initial state marks should be put within a loop to guarantee that the number of control steps required for each iteration of the loop is minimal (if performance is the optimization criterion). The following approach (this is effectively what has been implemented at present) gives good result (it seems to be optimal if there is a single path in the loop body though yet to be proved).

The traversing starts from the entry point of the loop (beginning of the loop body) towards the exit point (end of the body) using the main scheduling procedure (`FollowPath`) with minor modifications - clock period and resource constraints are removed to find out the best possible place for the state mark considering only storage cost and operations, which require synchronization. Inserting a state mark ensures that the loop has been broken and the procedure `FollowPath` can be used taking into account clock period and resource constraints.

5.6. Rules for state marking

As mentioned earlier, the state marks decide the boundaries of the FSM, but this needs to be done in a way to preserve the semantics implied by the *wait* marks. Accordingly, the state marking procedure ensures that on any control path, there is at least one state mark. The rules, illustrated in Figure 5.13., are as follows:

Rule 1: between any two consecutive *wait* marks (or coinciding with one of them);

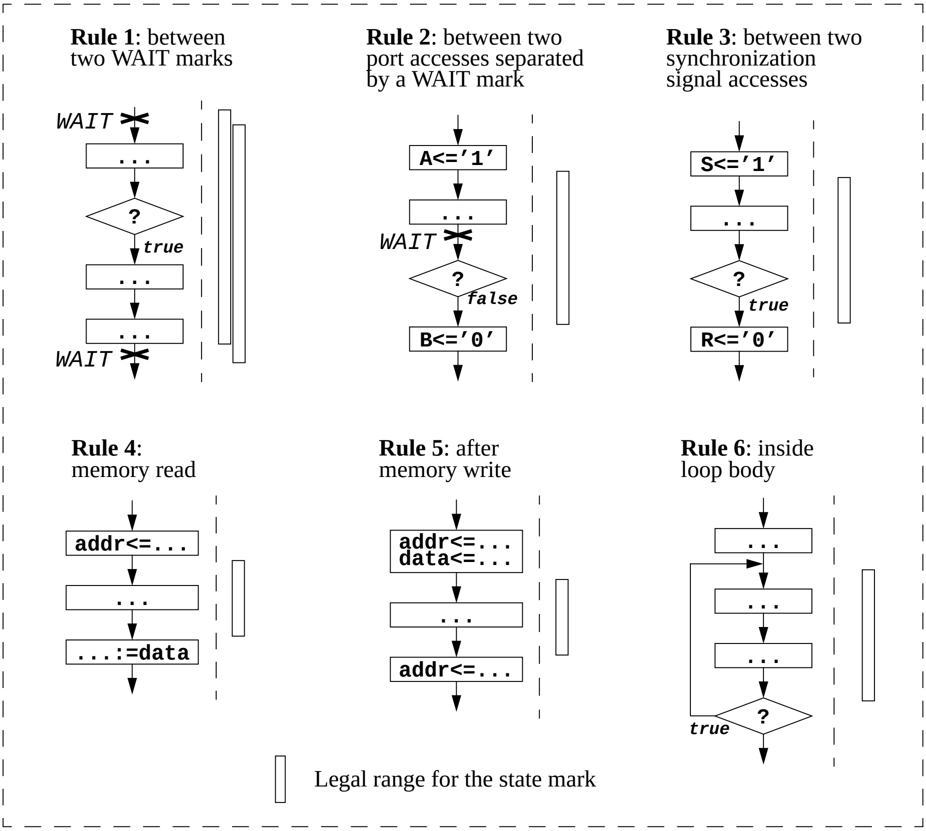
Rule 2: between any two consecutive port or signal accesses separated by a *wait* mark;

Rule 3: between any two consecutive accesses to the special ports/signals (synchronization between parallel threads and/or between master and slave controllers);

Rule 4: between sending address to a memory and reading data from that memory (this assumes that the address is latched; if the model is different, the rule is formulated accordingly);

Rule 5: between two consecutive accesses to the same memory;

Rule 6: between the beginning of a loop body and the end of the same loop body.



Rule 1: between two WAIT marks
WAIT
...
?
true
...
...
WAIT
Rule 2: between two port accesses separated by a WAIT mark
A<='1'
...
WAIT
?
false
B<='0'
Rule 3: between two synchronization signal accesses
S<='1'
...
?
true
R<='0'
Rule 4: memory read
addr<=...
...
...:=data
Rule 5: after memory write
addr<=...
data<=...
...
addr<=...
Rule 6: inside loop body
...
...
...
?
true
Legal range for the state mark

Figure 5.13. Rules for state marking

An intersection of state mark ranges is used when more than one rule is applied on a path segment. The use of the rules is illustrated later in Figure 5.14.

In addition to these rules, the clock period and resource constraints dictate insertion of state marks. The scheduling algorithm also checks that the delay of each *operation* node is less than length of the clock period. If this is not the case, the node is split and the operations distributed between the resulting nodes.

In contrast to the path-based scheduling, variables can be assigned more than once in one state. This assumption has been made to allow scheduling of complex decisions and conditional actions to be executed within a clock cycle. In addition, it is assumed that the number of arithmetic units is not a strong constraint because of the nature of the CMIST systems there are very few arithmetic operations, which can compete. If this is not the case, these units have to be locally rescheduled beforehand.

The scheduling procedures, described in section 5.4., are used as illustrated below. The scheduling is performed in three phases:

- First some of the memory reads are marked (accordingly to the *Rule 4*). This can be done as the first step because of the latched addresses; there is no freedom to select a place for the state mark between sending address to a memory and reading data from that memory.
- Detecting loops and scheduling them (see also section 5.5).
- Scheduling of all segments starting from the existing state marks; this creates sub-segments which are treated similarly.

The first two phases, described above, also guarantee splitting of the CFG into segments (acyclic sub-graphs) necessary for the last phase. The first phase marks edges between two consecutive operational nodes when an operation in the first node sends address to a memory and an operation in the second node reads data from that memory. In other words, the *Rule 4* is applied under simplified circumstances. The state marks *S3* and *S4* in Figure 5.2. has been generated during this phase. In the next two phases procedure `FollowPath` is used which traverses in recursive manner all path segments. `AnalyzePath` checks for the termination conditions (see rules described above) and decides the actual place of the state mark along the segment - the cheapest (number of storage elements needed) and/or the fastest (as late as possible) solution.

The other states marked (Figure 5.2.):

- *S5* - between two consecutive accesses to the same memory (Rule 5);
- *S6* - the same as *S5*, additionally priority of the *false* path was used to forced the marking;
- *S7* - AFAP schedule of the loop body (Rule 6).

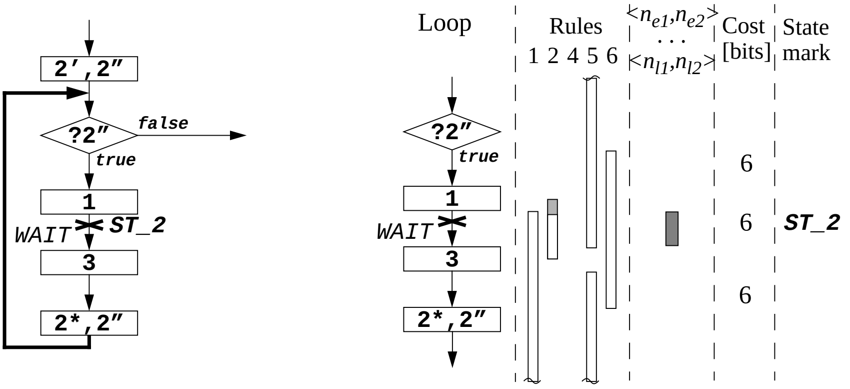
An example of a loop scheduling is shown in Figure 5.14. The resulting state marks is *ST_2*. No termination conditions were triggered when scheduling segments and therefore no additional states were generated. In Figure 5.14.c), from left to right are shown: the CFG segment under analysis; activated rules (Rules), where empty boxes correspond to the constraints and shadowed boxes show that a significant operation has been registered but no constraints applied yet; path segment for state marking ($<n_{e1},n_{e2}>,<n_{l1},n_{l2}>$); cost of a possible state marking in the number of bits; and the decided marking.

5.7. Results of experiments

To compare different control oriented scheduling strategies the example shown in Figure 5.2. was used. In all cases, the resulting Mealy FSM was also synthesized using AMS 0.8 micron CMOS 3.3 V technology. For resource constrained scheduling it was assumed that only one comparator and one adder were available. The synthesis results are shown in Table 5.1. and Table 5.2. where:

```
 . . .
 -- cell_var - buffer (53*8 bits)
 -- “array (0 to 52) of nat8”
 ack_in <= ‘0’;                      -- 1
 for i in 1 to 52 loop               -- 2
   wait on clk until clk=’1’;
   cell_var (i) := data_in;          -- 3
 end loop;
 . . .
```

a) VHDL code.
Complex operations divided into elementary ones:
2 - loop header - **2’**, **2”** and **2*** - initializing, checking and incrementing;



b) corresponding IRSYD

c) scheduling

Figure 5.14. An example of a loop scheduling.

- columns show different scheduling strategies;
- rows “states/transitions”, “registers/bits”, “area [gates]” and “delay [ns]” characterize result;
- rows “cycles” and “execution time [µs]” characterize performance of the result assuming that there were 10 elements in the array to be sorted.

The number of cycles (required to perform the sorting) was calculated as follows:

$cycles = ((N^2-N)/2)*(p_t*inner\_loop_t+p_f*inner\_loop_f) + (N-1)*outer\_loop + init$,

where N is number of variables (10), p_t and p_f - probabilities of swapping two variables (both 0.5), $inner\_loop_t$ and $inner\_loop_f$ - number of cycles of the inner loop, *outer_loop* - number of cycles to initialize the inner loop, and *init* is the number of cycles to initialize and conclude the sorting.

The state oriented scheduling gave identical results for both cases - the loops were not treated separately, and the loops were scheduled in the first order - and therefore only one column rep-

```
j := 1;                    -- 1
while j <= max_addr loop   -- 2
  i := max_addr;           -- 3
  while i >= j loop        -- 4
    w1 := mem(i-1);        -- 6
    w2 := mem(i);          -- 7
    if w1 > w2 then        -- 8
      mem(i-1) := w2;       -- 9
      mem(i) := w1;        -- 10
    end if;
  i := i-1;                -- 11
  end loop;
  j := j+1;                -- 12
end loop;
```

Variable **ix** is replaced with operation **i-1** (decrement).

Figure 5.15. Modified VHDL code.

resents the synthesis results.

The second table (Table 5.2.) represents results of synthesis of the code modification, which is based on assumption that an extra register can be much more expensive than an incrementer. The modified VHDL code is shown in Figure 5.15.

It can be seen that although the PBS and LDS gave shorter clock periods (“delay”), the resulting state machines were larger and the performance were worse compared to the proposed approach. The main reason is the lack of the operation chaining - more control steps and states were needed. The characteristics of the code - intensive memory access - did not allow LDS to improve scheduling compared with PBS, because the memory accesses forced the states in most of the cases.

The last column shows results from the prototype tool (see chapter 2), which partly implements the transition oriented scheduling approach (partly because CFG modifications are needed and these are not implemented currently). The scheduler estimates also cost of the components and therefore resource constraint scheduling was not performed because the components were not worth of reusing.

5.8. Conclusion

Scheduling is a very important task in HLS. Data flow oriented scheduling algorithms do not work well with control dominated systems since they are not designed to handle complex control constructs, loops and operator chaining efficiently.

A new scheduling method has been proposed for control dominated systems which shows su-

Table 5.1. Scheduling and synthesis results of Bubble sort example (variable “ix” is used).

		State-oriented	Transition-oriented		PBS	LDS	CMIST
			no loops	loops first			
resource constrained	states / transitions	7 / 10	7 / 12	7 / 12	8 / 10	8 / 10	-
	registers / bits	5 / 40	5 / 40	5 / 40	6 / 48	6 / 48	
	area [gates]	488	511	511	519	519	
	delay [ns]	20.3	21.5	21.5	17.4	17.4	
	inner_loop$_t$ / inner_loop$_f$	4 / 3	4 / 3	4 / 3	5 / 4	5 / 4	
	outer_loop / init	2 / 3	1 / 2	1 / 2	2 / 4	2 / 4	
	cycles	178.5	168.5	168.5	224.5	224.5	
	execution time [µs]	3.62	3.62	3.62	3.91	3.91	
resource unconstrained	states / transitions	7 / 12	7 / 13	7 / 16	8 / 10	8 / 10	7 / 11
	registers / bits	5 / 40	5 / 40	5 / 40	6 / 48	6 / 48	6 / 41
	area [gates]	511	504	499	519	519	518
	delay [ns]	21.5	22.9	24.5	17.4	17.4	25.0
	inner_loop$_t$ / inner_loop$_f$	4 / 3	4 / 2	4 / 2	5 / 4	5 / 4	4 / 2
	outer_loop / init	1 / 2	1 / 2	1 / 1	2 / 4	2 / 4	1 / 3
	cycles	168.5	146.0	145.0	224.5	224.5	147.0
	execution time [µs]	3.62	3.34	3.55	3.91	3.91	3.68

perior results compared to the commercial tools studied and considerably narrows the gap between highly optimized manual designs and automatically synthesized designs. The results are comparable with other control flow scheduling algorithms developed in recent years. The main difference is the capability to schedule complex logical operations together with some data flow operations within one clock cycle, very important when synthesizing high throughput CMIST systems.

Various data flow based scheduling techniques and data flow and control flow transformations are planned to add to the scheduler to allow both speculative execution and incremental improvements of the overall schedule. Potential path explosions, caused by deep nested if-statements, can be avoided by converting control flow constructs into data flow constructs. All these transformations not only would help to improve generated schedules, but also would widen the application area.

Table 5.2. Scheduling and synthesis results of Bubble sort example (variable “ix” is replaced with decrement operation).

		State-oriented	Transition-oriented		PBS	LDS	CMIST
			no loops	loops first			
resource constrained	states / transitions	7 / 10	7 / 12	7 / 12	8 / 10	8 / 10	-
	registers / bits	4 / 32	4 / 32	4 / 32	5 / 40	5 / 40	
	area [gates]	424	473	473	461	461	
	delay [ns]	22.2	22.7	22.7	16.6	16.6	
	inner_loop$_t$ / inner_loop$_f$	4 / 3	4 / 3	4 / 3	5 / 4	5 / 4	
	outer_loop / init	2 / 3	1 / 2	1 / 2	2 / 4	2 / 4	
	cycles	178.5	168.5	168.5	224.5	224.5	
	execution time [µs]	3.96	3.82	3.82	3.73	3.73	
resource unconstrained	states / transitions	7 / 12	7 / 13	7 / 16	8 / 10	8 / 10	7 / 11
	registers / bits	4 / 32	4 / 32	4 / 32	5 / 40	5 / 40	5 / 33
	area [gates]	473	486	488	461	461	503
	delay [ns]	22.7	22.5	25.3	16.6	16.6	25.0
	inner_loop$_t$ / inner_loop$_f$	4 / 3	4 / 2	4 / 2	5 / 4	5 / 4	4 / 2
	outer_loop / init	1 / 2	1 / 2	1 / 1	2 / 4	2 / 4	1 / 3
	cycles	168.5	146.0	145.0	224.5	224.5	147.0
	execution time [µs]	3.82	3.29	3.67	3.73	3.73	3.68

6. Unified Allocation and Binding

In this chapter, fast and simple heuristic algorithms for unified allocation and binding are described. These algorithms were designed to make the design flow of the CMIST tool, described in chapter 2, complete.

6.1. Introduction

Datapath synthesis is an important step in any High Level Synthesis (HLS) tool. Most of the datapath synthesis approaches break up the datapath synthesis into three steps, namely, scheduling, allocation and binding [GDW93], [DMi94]. Generally, behavioral description of an algorithm is translated into a Control and Data Flow Graph (CDFG) representation. Scheduling decides the time step at which an operation will be executed. Allocation decides the number and type of units required for datapath implementation; i.e. registers/memories for variables, functional units for arithmetic operations, and multiplexers/buses for interconnections. The binding step finally binds the variables to available registers/memories, operations to available functional units, and interconnections to available multiplexers/buses. Most of the tools implement these steps separately because of the computational complexity of corresponding tasks.

Generally, the binding task can be mapped onto one of the two graph problems - clique partitioning or graph coloring. If the task can be represented as a conflict graph then the binding can be looked onto as a graph coloring problem. If the task is represented as a compatibility graph, then the binding can be looked onto as a clique partitioning problem. Since a clique partitioning problem can be mapped onto a graph coloring problem, and vice versa, it is enough to focus onto coloring only. Weights can be added both to the nodes and to the edges to define parameters of operations and/or of technology. Optimal solutions to these graph problems require non-polynomial time and, therefore, are impractical. A large number of greedy algorithms have been proposed to find fast but non-optimal solutions to these problems [GDW93], [DMi94]. Recent works on coloring of real life graphs has shown that for a large subset of these graphs, graph coloring can be done optimally in polynomial time [Cou97], [KiPo98]. Unfortunately, both works with non-weighted graphs and therefore target mostly binding in software.

In most of the HLS approaches and systems, storage binding, functional unit binding and interconnection binding tasks are solved separately and sequentially. It has been realized that a unified binding, though harder, can lead to a better solution. In this chapter, fast and simple heuristic algorithms, which solve the allocation and binding problems in a unified manner, are described. The algorithms solve the graph coloring problem by working on a weighted conflict graph. The weights represent parameters of units, e.g. bit-width, type, cost, etc. Use of weights allows to solve allocation and binding tasks together, and to allocate/bind interconnections at the same time with functional units and/or storage elements. The algorithms differ in the heuristic strategies.

The algorithms are extensions of a previous work [EKH97] and have been incorporated into the CMIST HLS tool xTractor. In CMIST applications, there are very few arithmetic operations and, therefore, there is very little scope of reuse or sharing of functional units. In xTractor, allocation and binding of functional units is merged. The same applies to register allocation and binding. In both these merged allocation and binding steps, the cost of interconnections is estimated and used. A designer has a choice to carry out register allocation and binding separately or together. The performance of heuristic algorithms was compared using more than 150 different real life examples and 150 random examples.

The allocation and binding tasks have been studied by many authors and various solutions have been proposed. In most of the cases the allocation and binding have been mapped onto ILP, graph coloring or graph partitioning tasks. All these optimization tasks are intractable for realistic-size examples. Thus different heuristics, which generate solutions quickly without guaranteeing optimality, have widely been used. [Pau88], [Hem92], [GDW93], [DMi94], [Cou96], [Cou97], [EKP98], [KiPo98], [YuKu98]

The simplest constructive algorithms were chosen because the fast heuristics showed good performance - solutions close to the global optimum are generated in matter of seconds. These simple heuristics were incorporated into the allocation and binding step of the synthesis tool.

In the next section, the construction of the conflict graph for allocation and binding task is described. In section 6.3 the heuristic algorithms are given and the results are discussed in section 6.4. The concluding remarks are given in section 6.5.

6.2. Conflict graph construction

In this section, construction of the weighted conflict graph is described for both functional unit and register allocation and binding. The graph is constructed from a scheduled XFC; a special kind of CDFG representing a Mealy FSM, is used for a better encapsulation of the control flow of CMIST applications.

A conflict graph is an undirected graph $G=(N,E)$. The nodes $v \in V$ represent behavioral entities (variable, operation) and the edges give the conflict relation, i.e. $<n_i,n_{\mathrm{j}}> \in E$ if and only if n_i and n_j cannot share the same resource. All nodes are associated with a weight and type, which are used by the cost function. The weights are assigned during the graph construction and are equal to the width of the unit in bits. The types define, which kind of hardware models can be used to implement corresponding functional units. An add operation, for instance, can be implemented using not only adders but also adder-subtracters or ALU-s. For storage elements, a single reg-

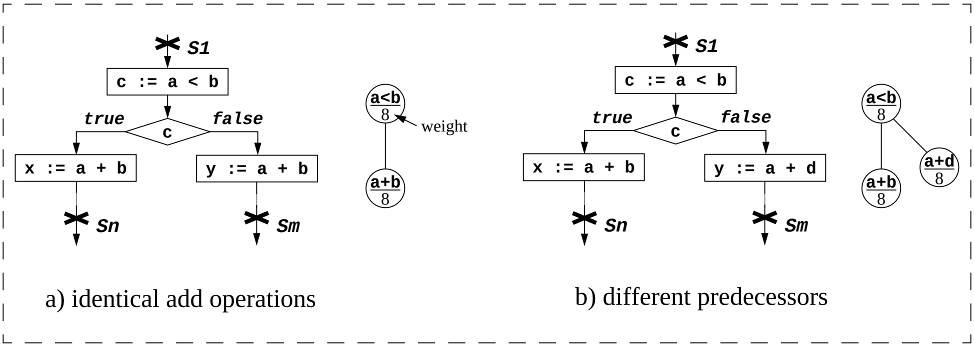


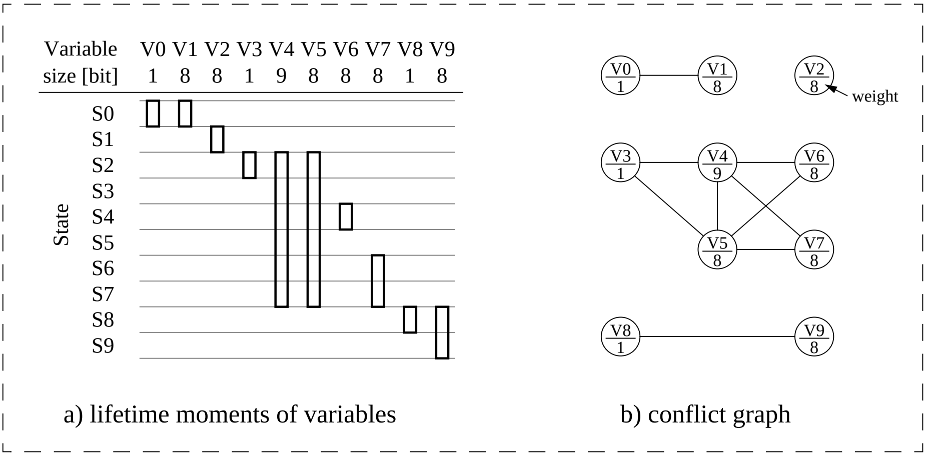
Figure 6.1. Parts of XFC-s and the corresponding parts of conflict graphs

ister type is used currently. Different register types can be used when, for instance, testability is incorporated as additional cost in the register allocation algorithm by [OlTi98].

The conflict graph for functional unit allocation and binding is constructed as follows. The FSM is analyzed transition by transition. As all transitions from a state in a Mealy FSM are mutually exclusive they can be analyzed independently. The operations, which can share functional units are collected together. Only arithmetic and relational operators are collected, since their sharing of resources can lead to reduction of overall cost. While collecting operations, their operands are also considered (hence the inputs to the resultant functional unit). This allows us to estimate the interconnection cost. All operations on the same transition require separate resources and, therefore, define conflicting operations of the graph. Two examples of conflict graph, one with identical add operations and another with different predecessors, are shown in Figure 6.1.

For register allocation, “lifetime moments” of variables are collected. The lifetime moments can be viewed as lifetime intervals but this can be misleading because of branching (conditions and loops). Only variables, which are assigned in one state but consumed in another state, are considered. For instance, in Figure 6.1., variables “a”, “b” and “d” are considered as register candidates; but variable “c” can be ignored if it is used only for branching, i.e. just after assignment. Variables with overlapping lifetimes define the conflicting nodes of the graph. An example of lifetime moments of an algorithm and corresponding conflict graph are shown in Figure 6.2.

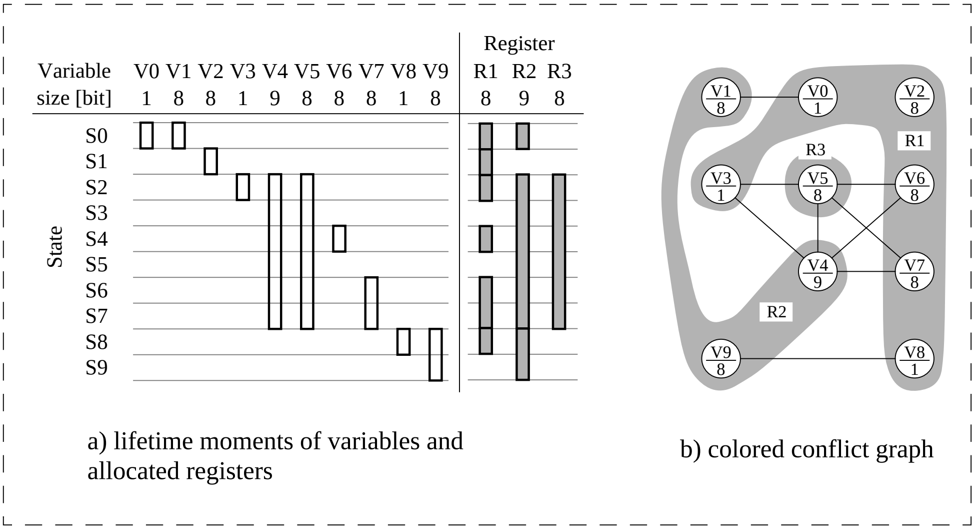
It is easy to see that to color a non-weighted graph, corresponding to the weighted graph in Figure 6.2., only three colors are needed, i.e. three registers. Let us consider the following coloring - “left-edge” (Figure 6.3.), as a result of the well known left-edge algorithm [GDW93], [DMi94] - *R1 = {V0, V2, V3, V6, V7, V8}*, *R2 = {V1, V4, V9}* and *R3 = {V5}*. This means that two 8-bit and one 9-bit registers are needed, and one 6x8-bit and one 3x9-bit multiplexers. At the same time only parts of the multiplexers are used because there is only one 9-bit variable (V4), and three of the variables have only one bit (V0, V1 and V8). Although the later optimization phases can reduce the size of multiplexers (interconnections) it is beneficial consider this already during the allocation and binding.



a) lifetime moments of variables

b) conflict graph

Figure 6.2. Lifetime moments of variables and corresponding conflict graph



a) lifetime moments of variables and allocated registers

b) colored conflict graph

Figure 6.3. Coloring result after applying the left-edge algorithm

To estimate interconnections, the cost function must take also into account how many candidates are mapped onto the same unit, i.e. how many nodes have the same coloring. The cost function of a color, modeled on the hardware implementation, is defined as follows:

$$C(t, w, n) = \begin{cases} C_U(t,w) & ;n < 2 \\ C_U(t,w) + m \cdot C_{MUX}(w, n) & ;n \geq 2 \end{cases}$$

where $C_U(t,w)$ is the cost of the hardware unit and $C_{MUX}(w,l)$ is the cost of the multiplexer (interconnections); t is the type of the color (type of the hardware unit - adder, subtracter, etc.), w is the largest weight of all nodes colored by that color (width of the register in bits for instance), m is the number of inputs of the unit (2 for adders, etc.) and l is the number of nodes colored by this color (number of inputs of the multiplexer). The cost function takes into account that there is no need for the multiplexer when only node is colored with the color; i.e. only one variable is allocated to the register or a single operation is allocated to a functional unit. The cost functions of units and multiplexers are modeled based on the actual hardware implementations and are represented as simple polynomials. For most of the units, except multipliers and multiplexers, simple linear functions give good approximations. When using LSI-10K technology, the cost functions for registers and multiplexers can be approximated as follows (result is in equivalent gates):

$$C_U(register, w) = 7.0w \qquad and \qquad C_{MUX}(w, n) = w \cdot (0.083n^2 + 1.49n + 0.154)$$

Cost functions and cost evaluations are constructed in a way which takes into account the number and type of the hardware modules needed. Figure 6.4. depicts the following two situations:

- one additions and one subtraction are mapped onto an adder-subtracter and two multiplexers, one for each operand; and
- two variables are mapped into a register and a multiplexer.

In both cases, one of the candidates has smaller bit-width thus creating a situation where some of the multiplexer inputs are not used. Sign extensions are not shown to avoid unnecessary details.

Based on the given cost functions, costs of the initial graph and the coloring, described above, can be calculated as follows:

- initial: C_{init} = 3*C_U(register,1)+6*C_U(register,8)+C_U(register,9) = 420 , and
- “left-edge”: $C_{left\text{-}edge}$ = C(register,8,6)+C(register,9,3)+C(register,8,1) = 320 .

It is worth of noting that although the “left-edge” is cheaper than the initial one, it is not the cheapest because of the non-optimal usage of multiplexers. The cheaper bindings are given in the next section as examples of heuristic colorings of the same graph.

The cost function is not limited to estimate area in gates. Additional costs, e.g. layout related, can be incorporated without changing the nature of the cost function and how it is used to calculate the cost of a coloring. Also, power consumption estimation can be used instead of area estimation, for instance, or various estimations can be combined. Various techniques can be used to estimate distance between nodes, and therefore also to estimate wiring [Pau88], [Hem92], [EKP98].

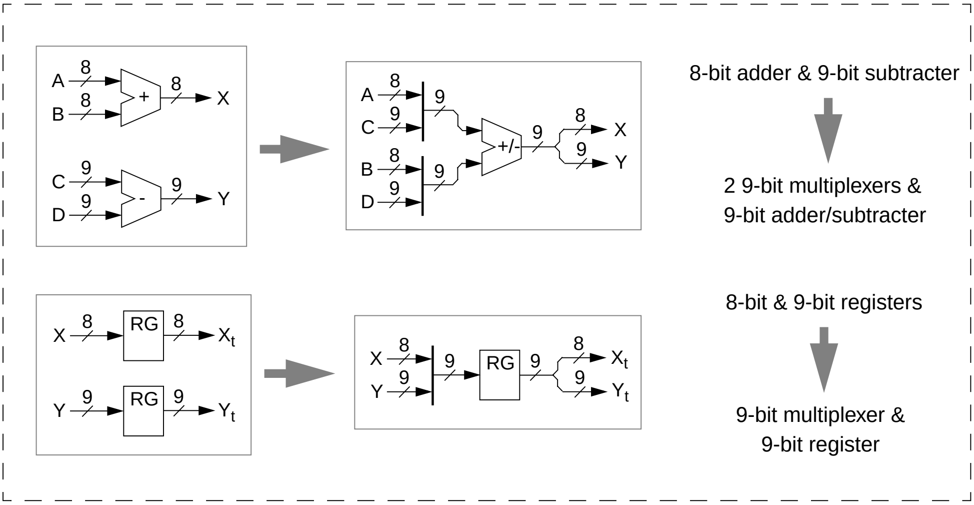


Figure 6.4. Binding examples

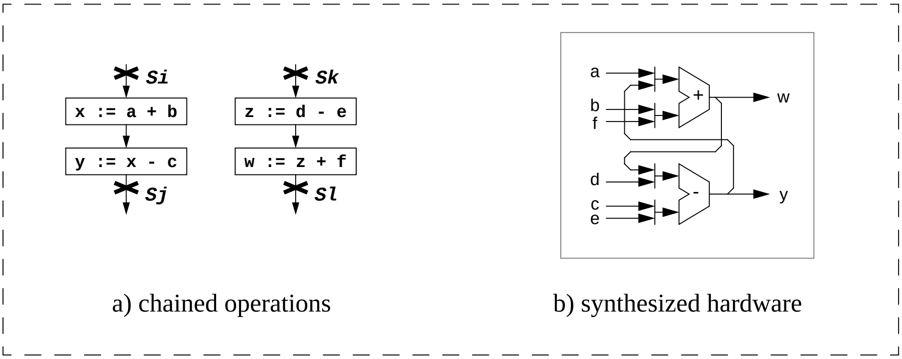


Figure 6.5. False loop caused by data path sharing

Reuse of chained operations creates a possibility for asynchronous feedbacks, called false loops, e.g. two reused arithmetic units are chained into a single clock step in different order in different states (see Figure 6.5). False loops can be caused also by sharing also in control path. This situation can be generated actually by any binding algorithm. Although the possibility for such a case is very low in CMIST style applications, and never occurred in practice, it should be taken into account anyway. A methodology, which detects and eliminates false loops, both from data path and control path, is proposed in [SLH96].

6.3. Heuristic algorithms

The heuristic algorithms, described below, are based on a greedy constructive approach in which colors are assigned to nodes sequentially. The greedy approach is the simplest amongst all the approaches, it is both easy to implement and computationally inexpensive. Unfortunately, it is liable to produce inferior results - as much as 100% off - because it takes into account locally available information only. [GDW93], [DMi94], [Cou97]

To overcome drawbacks of the greedy approach, additional heuristics are used. The following four different heuristics are used to define the selection order:

- the most expensive nodes first (H1);
- the cheapest nodes first (H2);
- random selection (H3, with multiple iterations);
- dynamic selection (H4) - the cheapest extra cost;
- branch-and-bound combined with dynamic selection (H5).

For all allocation and binding tasks, a node in the conflict graph corresponds to a behavioral entity (variable, operation), and a color corresponds to a resource (register, functional unit) used for its implementation. Cost of the existing coloring is evaluated at every step and the cheaper of two alternatives - using an existing color or creating a new one - is used. The cost evaluation also takes into account that some different operation types can be implemented using a single functional unit, e.g. operations ‘+’ and ‘-’ can be implemented as a shared adder-subtracter.

All five heuristics use the same core function (*BestBinding*, see Figure 6.6) to find the best coloring of a node. The function has two input parameters, partly colored graph (*G*) and index of the node (*node*) to be colored, and returns selected color (*best_color*) and type (*best_type*). The function is called at every iteration to evaluated the cost of the existing coloring and to select the cheapest of two alternatives - using an existing color or creating a new one. All five heuristics differ only how the nodes are ordered in the coloring process.

Smaller functions used in the function BestBinding and in the coloring procedures below are:

- *NextUncoloredNode* - selects a uncolored node from the graph (G) according to the second parameter which can have value costliest, cheapest, or random;
- *UniqueColor* - creates a new color;
- *CheapestType* - finds the cheapest type of the node;
- *Color* - associates the given node with the given color and type;
- *Cost* - calculates the cost of the graph (uncolored nodes are assumed to have unique colors);
- *CostColored* - calculates the cost of the graph (only colored nodes considered);
- *AllColors* - list all existing colors;

```
function BestBinding ( G, node ) begin
  (* A new color as reference *)
  best_color := UniqueColor(G);
  best_type := CheapestType(node);
  G' := Color(G,node,<best_color,best_type>);
  best_cost := Cost(G');
  (* All existing colors *)
  for  ∀ color ∈ AllColors(G)  loop begin
    (* Non-conflicting nodes only *)
    if  not EdgeConflict(color,node)  then begin
      (* Compatible node types only *)
      for  ∀ type ∈ CompatibleTypes(color,node)  loop begin
        G' := Color(G,node,<color,type>);
        cost := Cost(G');
        if  cost < best_cost  then begin
          best_color := color;
          best_type := type;
          best_cost := cost;
        end if;
      end loop;
    end if;
  end loop;
  return <best_color,best_type,best_cost>;
end function;
```

Figure 6.6. Function *BestBinding*

- *EdgeConflict* - returns true when there is a conflict between the given node (*node*) and any of the nodes colored by the given color;
- *CompatibleTypes* - returns a list of compatible types of the given node (*node*) and all of the nodes colored by the given color;
- *UncoloredNodes* - returns a list of all uncolored nodes.

The coloring procedures are as follows:

- The heuristics H1, H2 and H3 use exactly the same procedure (*StaticGreedy*, see Figure 6.7.). The only difference is in the selection order - the function *NextUncoloredNode* is called with an extra parameter, which defines the order - the largest first, the cheapest first and a pseudo-random order, respectively.
- The procedure for the heuristic H4 (*DynamicGreedy*) is similar to *StaticGreedy*, except that at every iteration a node is selected which gives the current best coloring (see Figure 6.8.);
- The procedure for the heuristic H5 (*BranchAndBound*, see Figure 6.9.) makes a recursive use of sub-procedure *BranchAndBoundStep*. In this way, all possible node selections are considered, although even this algorithm decides the color selection based on locally available information; i.e. not all possible type combinations are considered. The algorithm uses the heuristic H1 to find the initial boundary.

```
procedure StaticGreedy ( G, selection ) begin
  while  UncoloredNodes(G) ≠ ∅  loop begin
    node := NextUncoloredNode(G,selection);
    <best_color,best_type,best_cost> := BestBinding(G,node);
    G := Color( G, node, <best_color,best_type> );
  end loop;
  return G;
end procedure;
```

Figure 6.7. Procedure *StaticGreedy*

```
procedure DynamicGreedy ( G ) begin
  best_cost := oo;
  while  UncoloredNodes(G) ≠ ∅  loop begin
    for  ∀ node ∈ UncoloredNodes(G) loop begin
      <color,type,cost> := BestBinding(G,node);
      if  cost < best_cost  then begin
        G := Color( G, node, <color,type> );
        best_cost := cost;
      end if;
    end loop;
  end loop;
  return G;
end procedure;
```

Figure 6.8. Procedure *DynamicGreedy*

Assuming, that there are *n* nodes to be colored and there are no more than *t* types of functional units, the computational complexity of the heuristics is in the worst case as follows:

- heuristics H1, H2 and one iteration of H3 - $O(n^2 \cdot t)$;
- heuristic H4 - $O(n^3 \cdot t)$; and
- heuristic H5 - $O(n! \cdot t)$.

The actual number of iterations is smaller in practice since the complexity evaluations above do not take into account conflicts, which sometimes significantly reduce the number of possible colorings.

Figure 6.10. depicts some steps of a coloring example. The used heuristic is H1 (the largest nodes first), and the initial graph is the same as in Figure 6.2. Comparisons, i.e. the costs for different partial colorings, are shown at corresponding coloring steps. Cost of colors are shown

```
procedure BranchAndBound ( G ) begin
  best_G := StaticGreedy ( G, costliest );
  best_G := BranchAndBoundStep ( G, best_G );
  return best_G;
end procedure;

procedure BranchAndBoundStep ( G, best_G ) begin
  for  ∀ node ∈ UncoloredNodes(G)  loop begin        (* Branch *)
    <color,type,cost> := BestBinding(G,node);
    if  cost ≥ CostColored(best_G)  then  return;     (* Bound? *)
    Gx := Color( G, node, <color,type>);
    (* More branching? *)
    if  UncoloredNodes(Gx) ≠ ∅  then begin
      best_G := BranchAndBoundStep ( Gx, best_G );
    (* All colored *)
    else begin
      if  cost < CostColored(best_G)  then          (* The best? *)
        best_G := Gx;
      end if;
    end if;
  end loop;
  return best_G;
end procedure;
```

Figure 6.9. Procedure *BranchAndBound*

at every step in for $C_i(w,l)$, where C_i is the i-th color, w is the bit-with of the component and l is the number of multiplexer inputs. The type of the unit, *register*, is omitted since all units have the same type.

All heuristic colorings of the same conflict graph are shown in Figure 6.11. together with their cost and node selection order. Every color is marked with its cost $C(w,l)$, where the type of the unit is omitted again for simplicity. The best result was achieved when using the random selection, but it is important to note that the result depends on the initial seed of the pseudo-random number generator and therefore multiple runs of the H3 should be used. The branch-and-bound approach based algorithm (H5) gave also the best coloring, as it should be since for register allocation only one type is available, but with much greater execution time.

To illustrate how cost depends on the FU type let us consider a case where two additions and one subtraction are candidates for reuse. The available FU types are adders, subtracters and adder-subtracters. The cost functions are as follows:

- adder - $C(add,w) = 8.0 \cdot w - 3.5$;
- subtracter - $C(sub,w) = 9.0 \cdot w - 5.6$;
- adder-subtracter - $C(addsub,w) = 11.0 \cdot w - 1.9$;

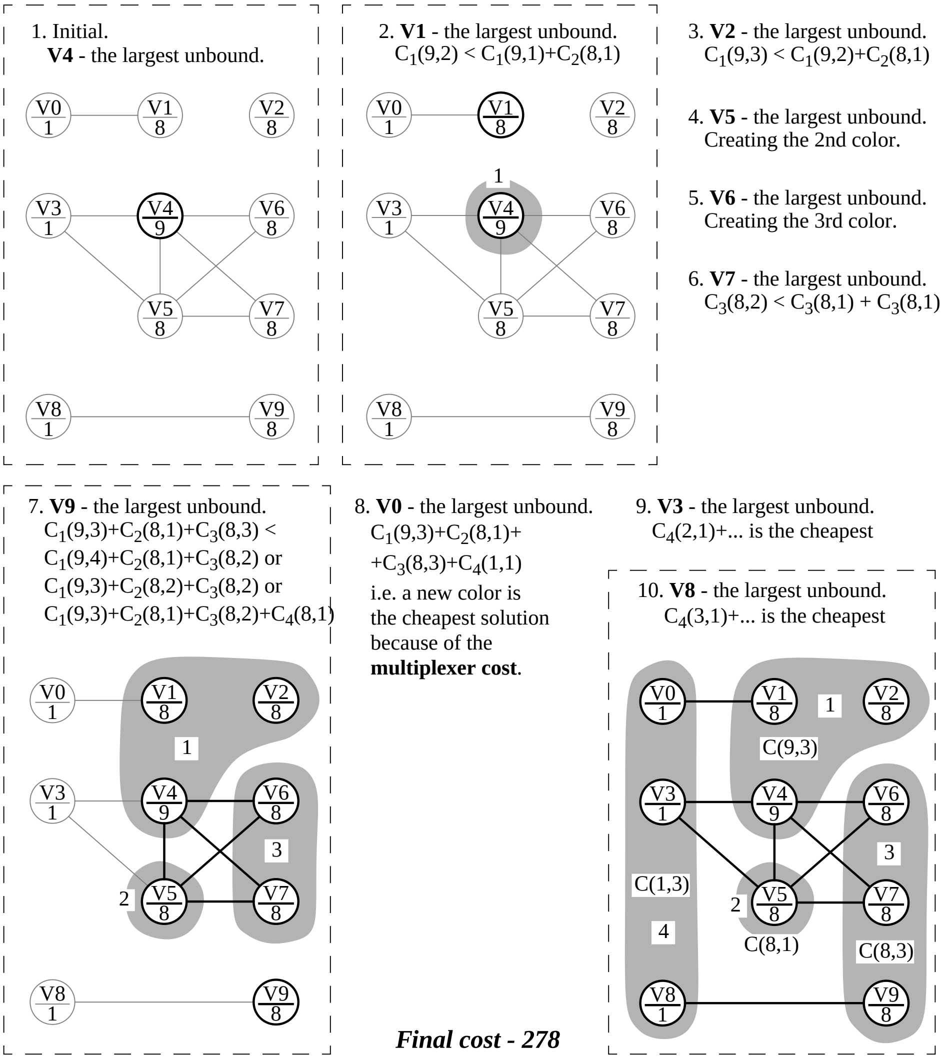

1. Initial.
V4 - the largest unbound.
2. V1 - the largest unbound.
C1(9,2) < C1(9,1)+C2(8,1)
3. V2 - the largest unbound.
C1(9,3) < C1(9,2)+C2(8,1)
4. V5 - the largest unbound.
Creating the 2nd color.
5. V6 - the largest unbound.
Creating the 3rd color.
6. V7 - the largest unbound.
C3(8,2) < C3(8,1) + C3(8,1)
7. V9 - the largest unbound.
C1(9,3)+C2(8,1)+C3(8,3) <
C1(9,4)+C2(8,1)+C3(8,2) or
C1(9,3)+C2(8,2)+C3(8,2) or
C1(9,3)+C2(8,1)+C3(8,2)+C4(8,1)
8. V0 - the largest unbound.
C1(9,3)+C2(8,1)+
+C3(8,3)+C4(1,1)
i.e. a new color is
the cheapest solution
because of the
multiplexer cost.
9. V3 - the largest unbound.
C4(2,1)+... is the cheapest
10. V8 - the largest unbound.
C4(3,1)+... is the cheapest
V0 1
V1 8
V2 8
V3 1
V4 9
V6 8
V5 8
V7 8
V8 1
V9 8
1
2
3
4
C(9,3)
C(1,3)
C(8,1)
C(8,3)
Final cost - 278

Figure 6.10. A step-by-step coloring example

- multiplexer - $C_{MUX}(w,l) = w\cdot(0.083^2+1.49\cdot l+0.154)$;

Table 6.1. lists possible bindings with bit-widths 4, 8 and 16 bits. The last two columns, where the reuse of functional units takes into account also the cost of multiplexers, show also improvement against the case when units are not reused. It is easy to see that for bit-width 4 the reuse gives very small improvement which means that when taking also into account wiring the 4-bit adders/subtracters are not worth of reusing.

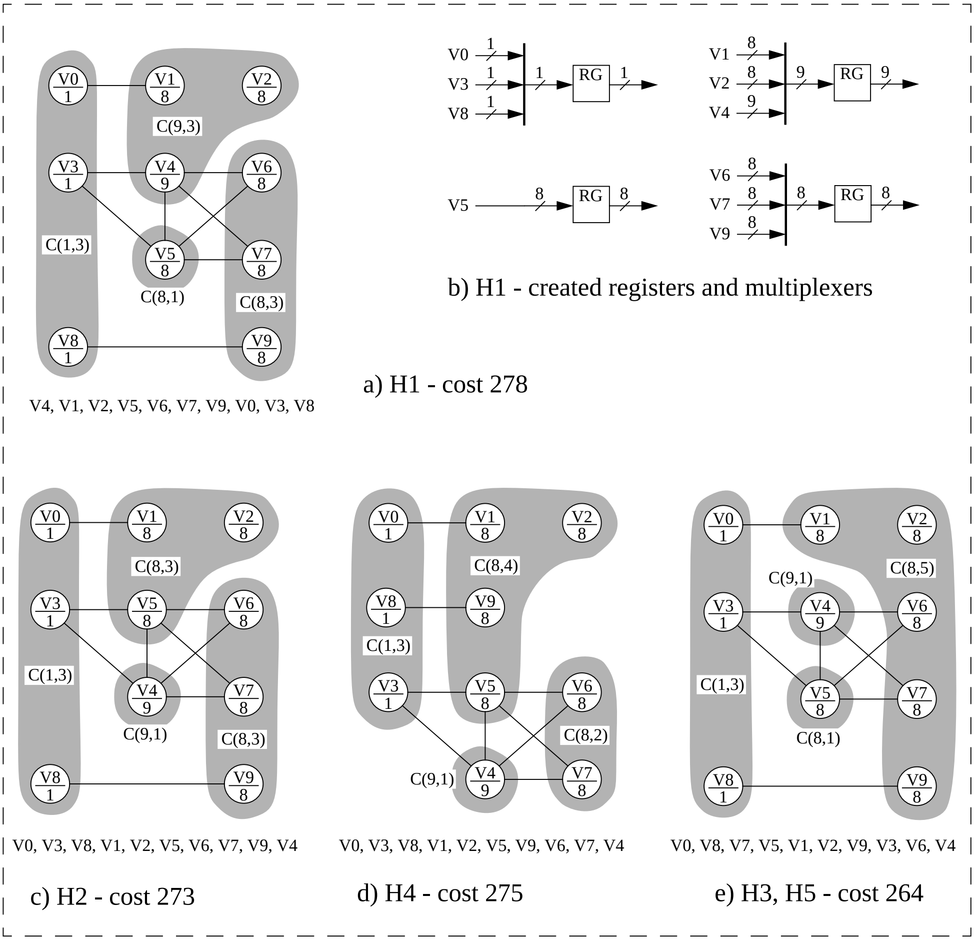

V0 1
V1 8
V2 8
C(9,3)
V3 1
V4 9
V6 8
C(1,3)
V5 8
V7 8
C(8,1)
C(8,3)
V8 1
V9 8
V4, V1, V2, V5, V6, V7, V9, V0, V3, V8
a) H1 - cost 278
RG
b) H1 - created registers and multiplexers
C(8,3)
C(9,1)
V0, V3, V8, V1, V2, V5, V6, V7, V9, V4
c) H2 - cost 273
C(8,4)
C(8,2)
V0, V3, V8, V1, V2, V5, V9, V6, V7, V4
d) H4 - cost 275
C(8,5)
V0, V8, V7, V5, V1, V2, V9, V3, V6, V4
e) H3, H5 - cost 264

Figure 6.11. Coloring results of different heuristics.

Table 6.1. Cost examples of FU reuse

Bit-width	**2 adders, 1 subtracter**	**1 adder, 1 subtracter**		**1 adder-subtracter**	
4	87.4	86.7	-0.8%	85.1	-2.6%
8	187.4	182.3	-2.7%	172.1	-8.2%
16	387.4	373.9	-3.5%	345.9	-10.7%

6.4. Results of experiments

More than 200 coloring tasks from real-life designs (from different bench mark example sources together with industrially relevant examples) were used when comparing the heuristic algorithms. The largest designs had 222 functional units which were bound into 92 components (estimated cost reduction from 42 kgates to 17 kgates), and 91 variables bound into 29 registers (cost reduction from 4 kgates to 2 kgates). The smallest designs had less than 10 units to be bound. It should be noted that the algorithm H3 was used with multiple trials and the best of the bindings was selected. The number of trials was set to the number of nodes of the graph, but in many cases the best result or one of the best, was achieved with the first trial.

The initial conflict graphs, constructed from the CDFG (XFC), do not take into account that operations with very different bit-widths are not worth of sharing the same hardware unit. The extra cost comes from wasted interconnections (unused inputs of multiplexers, for instance). All five heuristics were run also with a modification, which takes this into account, and constructs extra conflict edges between nodes which have too different costs. An extra conflict, i.e. an edge, is constructed when $C(type,max(w_{n1},w_{n2}),2) > C(type,w_{n1},1)+C(type,w_{n2},1)$, where w_{n1} and w_{n2} are the weights of the nodes.

Only for 78 designs (out of 246), the results from different heuristics were different (and different from the global optimum). For the rest of the designs, all heuristic algorithms obtained the optimum. Comparisons are presented in Table 6.2.

Table 6.2. Efficiency of different heuristics

Node selection heuristic		Best (of 58)	Minimal (of 58)	Average penalty	Maximum penalty	Average time
No extra edges	H1 (largest)	25	23	3.8 %	12.0 %	7.6”
	H2 (smallest)	23	14	3.8 %	9.2 %	4.5”
	H3 (random)	63	47	1.8 %	4.0 %	55.9”
	H4 (dynamic)	12	9	4.8 %	15.2 %	374”
Extra edges used	H1 (largest)	31	18	2.8 %	8.0 %	2.8”
	H2 (smallest)	21	13	3.8 %	10.4 %	2.9”
	H3 (random)	73	49	1.2 %	5.6 %	31.5”
	H4 (dynamic)	10	5	4.1 %	13.3 %	185”

The column “Best” shows how many of the bindings by that algorithm were also the overall best ones. The column “Minimal” shows how many of the bindings were also globally minimal. The two following columns, “Average penalty” and “Maximum penalty”, show how many percent the cost of the resulted bindings differ from the global minimum. The last column, “Average time”, shows the average coloring time of real-life graphs with 50 to 100 nodes. This column is added to show how the extra edges speed-up the optimization.

150 random graphs (from 10 to 50 nodes) were colored using the same four heuristics. The results were comparable with the coloring results of real-life graphs - the average penalty was 1% for H3 and 3-4% for other heuristics, and the maximum penalty was, correspondingly 3% and 10-12%.

The execution times on Sun UltraSparc work-station were less than one second for most of the designs - graph building time included. Even the largest designs - with node count from 50 to 100 - required only 5 to 10 seconds to execute the static greedy heuristics (H1 and H2). The most time-consuming was, of course, the heuristic, which combined branch-and-bound and dynamic greedy approach (H5). It should be noted that for designs with nodes more than approximately 15 the exact solution is not known because of the combinatorial complexity. For large designs the best known solution is used, which is very probably also the best solution or very close to it. The best known solutions were obtained running heuristics H3 and H5 with CPU time limit set to one hour.

It is usually assumed that dynamic greedy algorithms (algorithm H4) are the most efficient because of the continuous re-evaluation of local minima [GDW93], [DMi94]. The only way to explain the inefficiency of the algorithm H4 is that any of the globally wrong decisions will inevitably lead away from the global optimum, regardless of the local optimality of that decision. On the other hand, the efficiency of the multiple random selections (algorithm H3) can be explained by taking into account that there can exist many global optima. Additionally, any of the partial solutions which does not lead away from the most optimal can lead to the desired optimum.

6.5. Conclusion

In this chapter efficient unified allocation and binding heuristic algorithms for HLS of CMIST applications were presented. The allocation and binding task has been mapped onto weighted graph coloring which takes into account both the functional unit cost (or storage cost) and the interconnection cost. Based on the results from experiments with more than 200 real-life and 150 random weighted coloring tasks, it can be stated that the heuristics H1, H2 and H3 will lead to good enough results in a matter of seconds. In a significant number of cases, it may also lead to the global optimum. The slower heuristics (H4 and H5) can be speed-up significantly when the conflict graph construction takes also into account that nodes with too different weights are not worth of sharing the same resource and therefore effectively should be conflicting nodes.

The short design times allow also to explore potential solutions in more details, or to spend more effort in control synthesis. In future, the algorithms will be modified to keep track on multiple local optima thus significantly reducing probable dangers created by a single wrong decision. The same basic algorithm is used for allocation and binding of all three types of units - functional, storage and interconnection. This helps us to merge these three tasks into a single design step, and therefore allowing unified calculation of implementation cost.

7. Synthesis Examples

Data field pre-packing, segment-based scheduling and unified allocation/binding methodologies were introduced in the chapters 4, 5 and 6. In this chapter the results of applying these methodologies onto some industrially relevant test cases are presented. Some points of interest are discussed in detail.

7.1. Test case: OAM part of the ATM switch

A telecommunication circuit, which is a part of an ATM system, was selected to test synthesis methodologies presented in the previous chapters. The ATM (Asynchronous Transfer Mode) is a standard and the preferred method in dealing with increasing data rates in telecommunication systems. The ATM switch is an example from the CMIST application domain. The OAM (Operation and Maintenance), a part of the ATM protocol, is used as test case because it is a typical CMIST and non-trivial in its complexity. OAM is responsible for detection and reporting of errors and performance degradation in the ATM network at switch level. The name F4/F5 refers to the levels of OAM functionally covered by the block. The F4 level handles the OAM functionality concerning Virtual Paths (VPs) and the F5 level handles the OAM functionality concerning Virtual Channels (VCs). The levels F1 to F3 handle the OAM functionality at physical layer. Figure 7.1. depicts an ATM node and the location of F4/F5 block in it.

The F4/F5 block has been used as a test case to validate the initial CMIST synthesis approach ([HSE95], [SEP96]). A behavioral VHDL model of the F4/F5 block was developed at CADlab of Linköping University, Sweden [DHE94], and the test bench for the model was developed at ESDlab of Royal Institute of Technology, Sweden [Sva96]. The functional structure of the model is illustrated in Figure 7.1.

The initial functional model of the F4/F5 block used transaction based interfacing mechanism to synchronize data transmissions between modules of the block. There does not exist direct hardware equivalents of transactions and handshaking signals were introduced to synchronize the data transmissions between the block and the external world and between the component modules of the block.

The simulatable model was fully serial and could handle only one cell at a time. A partitioning, described below, was introduced to allow the parallel handling of multiple cells thus reducing performance requirements of component modules. Another reason for partitioning, and code modifications, was to make the model synthesizable by HLS tools. The initial model used par-

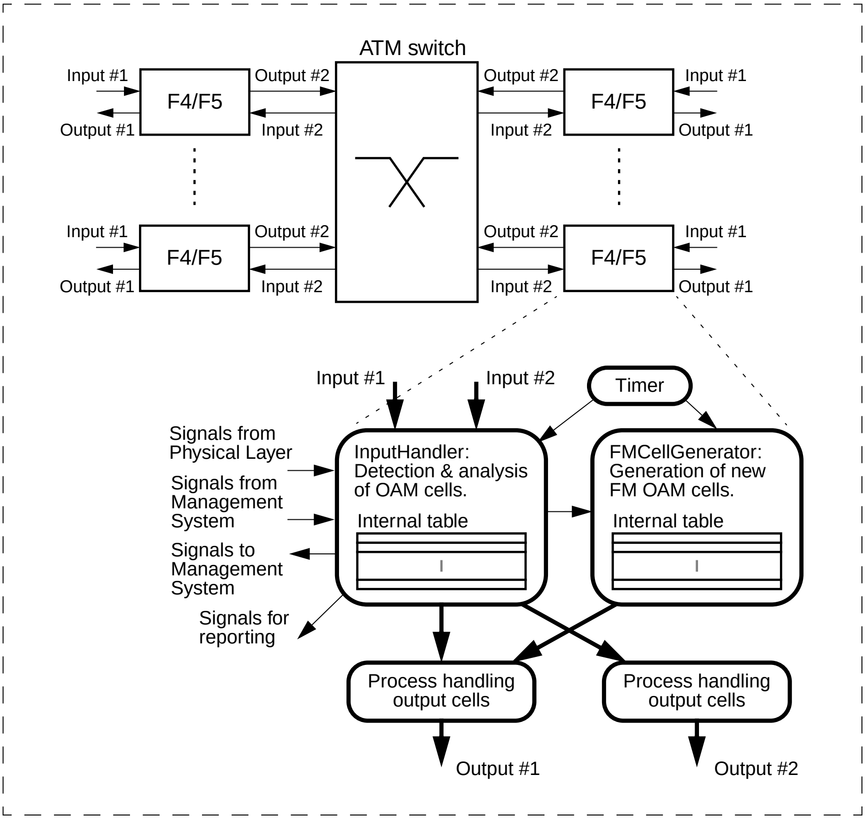


Figure 7.1. ATM node and OAM (F4/F5) block

allel representation of ATM cells - 424 bits each. Although the parallel representation is very comfortable for modeling it causes a lot of trouble in synthesis. Any implementation will have very tense wiring because all modules are connected through 424 bit wide buses. Additional difficulties for the synthesis tools were caused by the high complexity of the algorithm. The structure of the partitioned OAM block is shown in Figure 7.2.

The following two strategies were used to reduce the complexity of component modules:

- Data transmissions were serialized where possible thus reducing the wiring costs. This decreased the overall complexity, although introduced extra latencies.
- Analysis of data dependencies between the modules allowed to partition the initial algorithm of the input handler into smaller, less interdependent pieces. These parts could be handled by the synthesis tools more flexibly. Although the partitioning introduced some duplicated parts and increased the total area of the block, the overall performance was improved because the partitions could work independently thus increasing the throughput of the block.

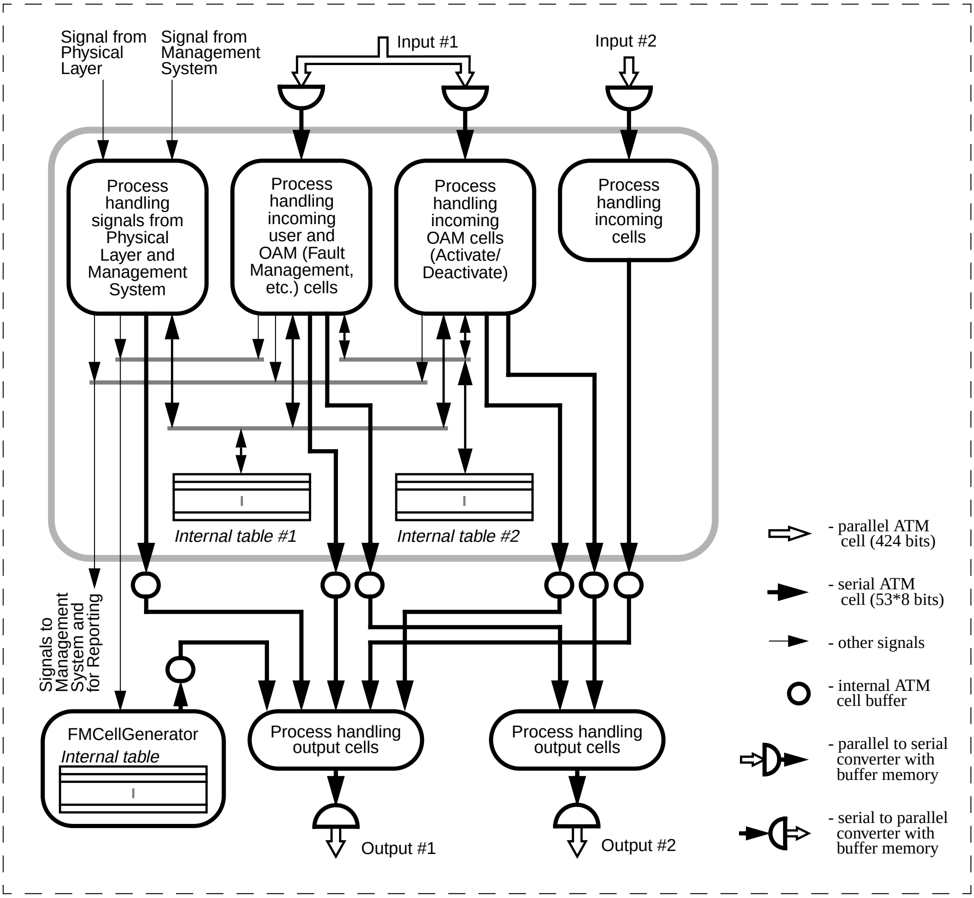

Signal from Physical Layer
Signal from Management System
Input #1
Input #2
Process handling signals from Physical Layer and Management System
Process handling incoming user and OAM (Fault Management, etc.) cells
Process handling incoming OAM cells (Activate/ Deactivate)
Process handling incoming cells
Internal table #1
Internal table #2
Signals to Management System and for Reporting
FMCellGenerator
Internal table
Process handling output cells
Process handling output cells
Output #1
Output #2
- parallel ATM cell (424 bits)
- serial ATM cell (53*8 bits)
- other signals
- internal ATM cell buffer
- parallel to serial converter with buffer memory
- serial to parallel converter with buffer memory

Figure 7.2. Partitioned OAM (F4/F5) block of ATM node

Cells are transmitted byte-by-byte between modules after serializing. Internal cell buffers were introduced to reduce the idle time, spent for transmissions by the handlers, and to avoid possible deadlocks. Although each of the internal buffers can store only one ATM cell it frees the sending handler to perform next operations, e.g. to receive the next cell.

The input handler was partitioned in three steps. First, an independent part, which deals only with the cells arriving from the input number 2, was extracted as “Input Handler #2”. Since the rest of the main input handler was still too large to be synthesized, an additional partitioning was needed. The functionality of the handler dictated two parts - one to handle cells from input channel 1 and another to receive signals from the Physical Layer and Management System (“Input PhL-MS”). A part of the channel status table, shared by both of the parts, was also extracted (“Internal table #1”).

The remaining handler was still too complex. The third cut took into account that the algorithm of the handler is actually a large case-statement. The resulting two parts, roughly of equal size, receive a copy of the cell from input channel 1 at the same time, process it or discard depending on the type of the cell. The first part (“Input Handler 1.1”) handles all user cells and OAM cells but Activate/Deactivate, and the second part (“Input Handler 1.2”) handles OAM Activate/Deactivate cells. Again, a part of the channel status table was extracted (“Internal table #2”).

The interfaces of the internal tables are using handshaking mechanism and they can handle one request at a time, the others must wait. A simple priority mechanism was introduced based on the assumed frequency of the requests.

The partitioned model was validated by using the test bench, which was developed for the initial simulatable model.

7.2. F4/F5: Applying data field pre-packing

The channel status tables of the F4/F5 block were good test cases to evaluate the efficiency of the data field pre-packing methodology. The entries of the tables consist of multiple data fields. These entries are accessed depending on the data received from the input channels or from the Physical Layer and Management System. The access patterns therefore can not be extracted from the code only and memory optimization techniques, which work well on voice and image processing applications (see e.g. [VLV95], [CWD98]), can not be used.

There exist five groups (structures) of data fields in the partitioned F4/F5 block which indexing is data depending (see also Table 7.1., columns “Group” and “Field”):

- IH11 - a part of the channel status, stored in the module “Input Handler 1.1”;
- IHPL - a part of the channel status, stored in the module “Input PhL-MS”;
- TBL1 - a part of the channel status, stored in the shared table “Internal Table #1”;
- TBL2 - a part of the channel status, stored in the shared table “Internal Table #2”; and
- FMCG - status of Forward Monitoring OAM cells, stored in a single table of the module “FM Cell Generator”.

Eight 1-bit fields in group TBL1 were packed intuitively into a single field (“s.mode”) when the F4/F5 was partitioned. This allowed a significant reduce in the number of memory accesses. Intuitive packing and some optimization of memory accesses, e.g. removal of duplicated accessed, reduced the total number of memory accesses from 219 in the simulatable F4/F5 model to 76 in the partitioned model. The number of read and writes accesses for different fields is shown in Table 7.1. - columns “Initial, r/w” and “Intuitive packing, r/w”.

The arrays, representing basic groups (BG) in the terms of the methodology presented in chapter 4, were allocated into 10 physical memories. It was assumed that the available memories can be

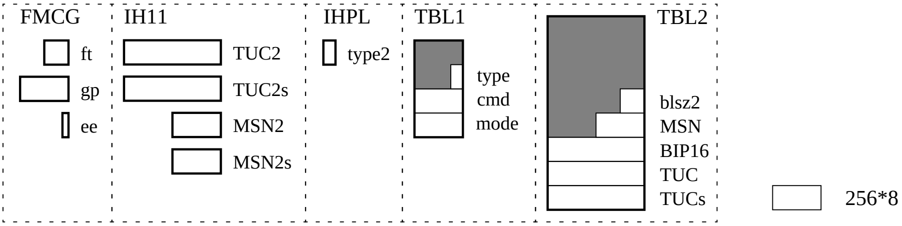


Figure 7.3. Memory allocation after intuitive packing

1, 2, 4, 8, 16 or 32 bits wide; and can store 256, 512, 1024 or 2048 words. The relative area and power consumption of each memory is shown in columns “Intuitive packing, area” and “Intuitive packing, power” (Table 7.1), respectively. The area and power consumption were normalized for later comparisons against other allocations. The layouts of data fields in the allocated memories are shown in Figure 7.3.

The data field pre-packing methodology was initially applied onto separate data field groups. The memory allocation resulted in two different packings, “Pre-packing #1” and “Pre-packing #2”, are presented in Table 7.1. These cases differ only in the packing of fields “s.MSN2” and “s.MSN2s” in group IH11. The total area remained the same but power consumption was reduced by 1%. A detailed analysis of pre-packing applied onto each of the groups is presented below.

Group IH11: The pre-packing of this group was an interesting case because of the complexity of data dependencies. The simplified code, shown in Figure 7.4., does not correspond exactly to the case “Intuitive packing” since some additional access optimizations have been performed. The data structure consists of four fields (BGs) - MSN2, MSN2s, TUC2 and TUC2s. Corresponding pruned CDFG is shown in Figure 7.5. *r(MSN2)* and *w(MSN2)* mark reading and writing the field MSN2, etc. Edges with circles denote control dependencies in the CDFG.

The table of dependency cases and corresponding compatibility graph are shown in Figure 7.6. Column “No.” in the table shows the total number of corresponding dependency cases in the CDFG. For clarity, the nodes of the compatibility graph are placed in such a way that their “read-sides” are directed to the left. Every node is marked with its field name, bit-width and with the number of read and write accesses. Every edge is marked with its weight, i.e. with the number of dependency cases.

Figure 7.7. depicts the result of the first clustering step of the compatibility graph from Figure 7.6. Both the new CDFG and the new compatibility graph are shown. Merging fields TUC2 and TUC2s reduced the number of memory accesses from 13 (6 reads and 7 writes) to 10 (5 reads and 5 writes). The simplified source code, corresponding to the packing, is shown in Figure 7.8. This packing corresponds to the case “Pre-packing #1” in Table 7.1.

```
typedef struct {
  unsigned TUC2: 16;    unsigned TUC2s: 16;
  unsigned MSN2: 8;     unsigned MSN2s: 8;
} InputHandler_1_1_record;

/*****  OAM cells  *****/
if (...) {
  p1->TUC2  = bits(cell,56,71);                 // w(TUC2)
  p1->MSN2  = bits(cell,48,55);                 // w(MSN2)
  p1->TUC2s = bits(cell,56,71);                 // w(TUC2s)
}
// ...
if ( p1->MSN2==bits(cell,48,55) ||              // r(MSN2)
     p1->MSN2s==bits(cell,48,55) ) {            // r(MSN2s)
  ... = p1->TUC2 - ... ;                        // r(TUC2)
}

/*****  user cells  *****/
new_TUC2 = p2->TUC2 + 1;                        // r(TUC2)
p2->TUC2 = new_TUC2;                            // w(TUC2)
if ( new_TUC2 - p2->TUC2s == 128 ) {            // r(TUC2s)
  p2->TUC2s = new_TUC2;                         // w(TUC2s)
  old_MSN2  = p2->MSN2;                         // r(MSN2)
  p2->MSN2s = old_MSN2;                         // w(MSN2s)
  p2->MSN2  = old_MSN2 + 1;                     // w(MSN2)
}
```

Figure 7.4. Simplified source code of IH11

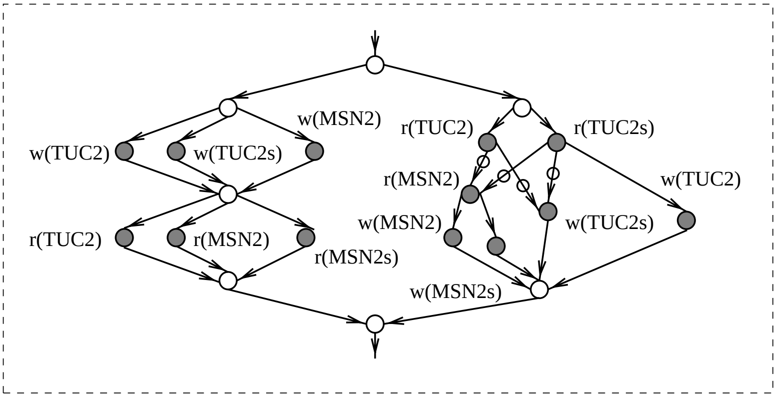
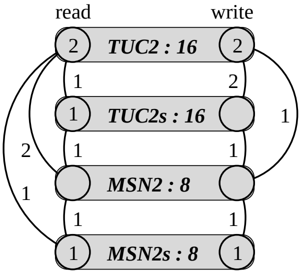


Figure 7.5. Pruned CDFG of IH11

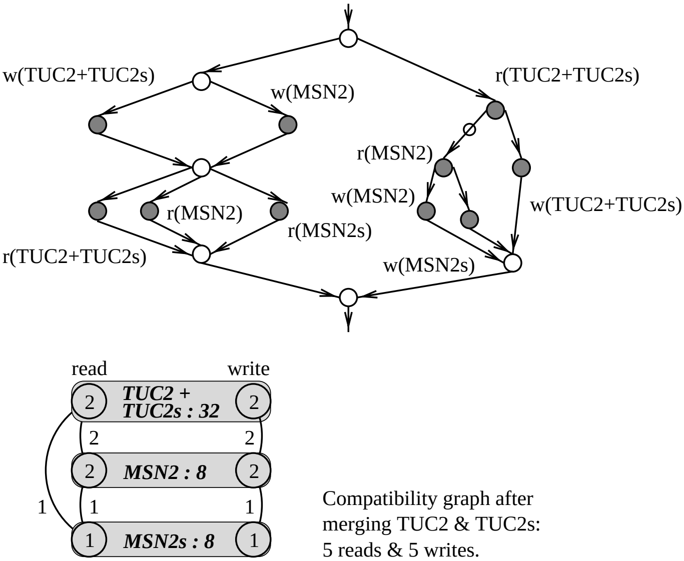
The second clustering step merged fields MSN2 and MSN2s. It should be noted that the merging of these fields introduced an extra read to avoid corruption of field MSN2 when writing MSN2s. At the same time the total number of accesses was reduced to 9 (5 reads and 4 writes). Figure 7.9. shows the resulting CDFG (the extra read is underlined) and the modified part of source code is shown in Figure 7.10. This packing corresponds to the case “Pre-packing #2” in Table 7.1.

Case	Fields	No.
read-read	TUC2-TUC2s	1
read-read	TUC2-MSN2	2
read-read	TUC2-MSN2s	1
read-read	TUC2s-MSN2	1
read-read	MSN2-MSN2s	1
write-write	TUC2-TUC2s	2
write-write	TUC2-MSN2	1
write-write	TUC2s-MSN2	1
write-write	MSN2-MSN2s	1



Compatibility graph:
6 reads & 7 writes.

Figure 7.6. Compatibility graph of IH11



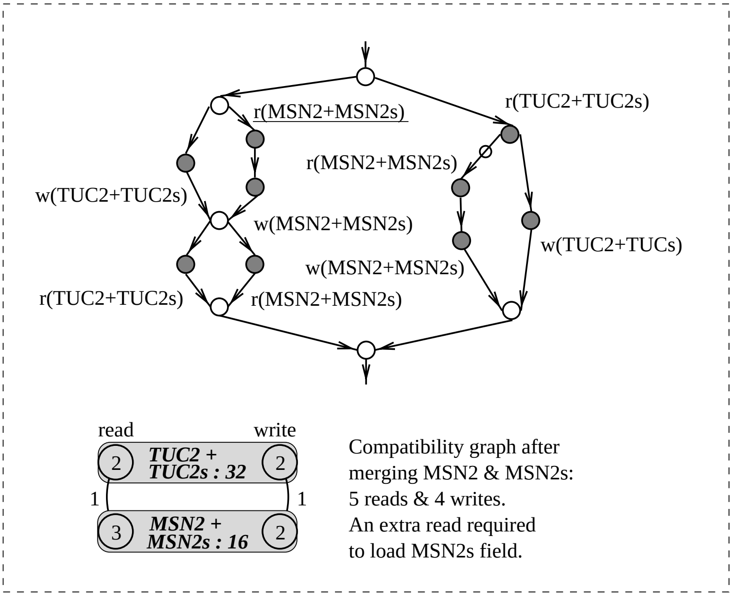
Compatibility graph after
merging TUC2 & TUC2s:
5 reads & 5 writes.

Figure 7.7. Clustering IH11 - step 1 (pre-packing #1)

Group FMCG was another interesting case since it pointed out that there exist cases when the compatibility graph clustering over-estimates the number of extra reads. Based on the analysis of this group the counter example, presented in section 4.7., was developed. Simplified source

```
typedef struct {
  unsigned TUC2_TUC2s: 32;
  unsigned MSN2: 8;     unsigned MSN2s: 8;
} InputHandler_1_1_record;

/*****  OAM cells  *****/
if (...) {
  p1->TUC2_TUC2s =                                  // w(TUC2+TUC2s)
    set_TUC2_TUC2s( bits(cell,56,71), bits(cell,56,71) );
  p1->MSN2       = bits(cell,48,55);                // w(MSN2)
}
// ...
if ( p1->MSN2==bits(cell,48,55) ||                 // r(MSN2)
     p1->MSN2s==bits(cell,48,55) ) {               // r(MSN2s)
  tmp_TUC2 = get_TUC2(p1->TUC2_TUC2s);             // r(TUC2+TUC2s)
  ...      = tmp_TUC2 - ... ;
}

/*****  user cells  *****/
tmp_TUC2_TUC2s = p2->TUC2_TUC2s;                   // r(TUC2+TUC2s)
new_TUC2       = get_TUC2(tmp_TUC2_TUC2s) + 1;
new_TUC2s      = get_TUC2s(tmp_TUC2_TUC2s);
if ( new_TUC2 - new_TUC2s == 128 ) {
  new_TUC2s = new_TUC2;
  old_MSN2  = p2->MSN2;                            // r(MSN2)
  p2->MSN2s = old_MSN2;                            // w(MSN2s)
  p2->MSN2  = old_MSN2 + 1;                        // w(MSN2)
}
p2->TUC2_TUC2s = set_TUC2_TUC2s(new_TUC2,new_TUC2s);// w(TUC2+TUC2s)
```

Figure 7.8. Simplified source code of IH11 (pre-packing #1)



Compatibility graph after
merging MSN2 & MSN2s:
5 reads & 4 writes.
An extra read required
to load MSN2s field.

Figure 7.9. Clustering IH11 - step 2 (pre-packing #2)

```
typedef struct {
  unsigned TUC2_TUC2s: 32;
  unsigned MSN2_MSN2s: 16;
} InputHandler_1_1_record;

/*****  OAM cells  *****/
if (...) {
  p1->TUC2_TUC2s =                                     // w(TUC2+TUC2s)
    set_TUC2_TUC2s( bits(cell,56,71), bits(cell,56,71) );
  old_MSN2s      = get_MSN2s(p1->MSN2_MSN2s);          // r(MSN2+MSN2s)
  p1->MSN2_MSN2s =                                     // w(MSN2+MSN2s)
    set_MSN2_MSN2s( bits(cell,48,55), old_MSN2s );
}
// ...
tmp_MSN2_MSN2s = p1->MSN2_MSN2s;                       // r(MSN2+MSN2s)
if ( get_MSN2(tmp_MSN2_MSN2s)==bits(cell,48,55) ||
     get_MSN2s(tmp_MSN2_MSN2s)==bits(cell,48,55) ) {
  tmp_TUC2 = get_TUC2(p1->TUC2_TUC2s);                 // r(TUC2+TUC2s)
  ...      = tmp_TUC2 - ... ;
}

/*****  user cells  *****/
tmp_TUC2_TUC2s = p2->TUC2_TUC2s;                       // r(TUC2+TUC2s)
new_TUC2       = get_TUC2(tmp_TUC2_TUC2s) + 1;
new_TUC2s      = get_TUC2s(tmp_TUC2_TUC2s);
if ( new_TUC2 - new_TUC2s == 128 ) {
  new_TUC2s = new_TUC2;
  old_MSN2  = get_MSN2(p2->MSN2_MSN2s);                // r(MSN2+MSN2s)
  p2->MSN2_MSN2s = set_MSN2_MSN2s(old_MSN2+1,old_MSN2 );// w(MSN2+MSN2s)
}
p2->TUC2_TUC2s = set_TUC2_TUC2s( new_TUC2, new_TUC2s ); // w(TUC2+TUC2s)
```

Figure 7.10. Simplified source code of IH11 (pre-packing #2)

code, presenting the dependencies between memory accesses, is shown in Figure 7.11. Corresponding pruned CDFG, table of dependency cases and compatibility graphs are shown in Figure 7.12.

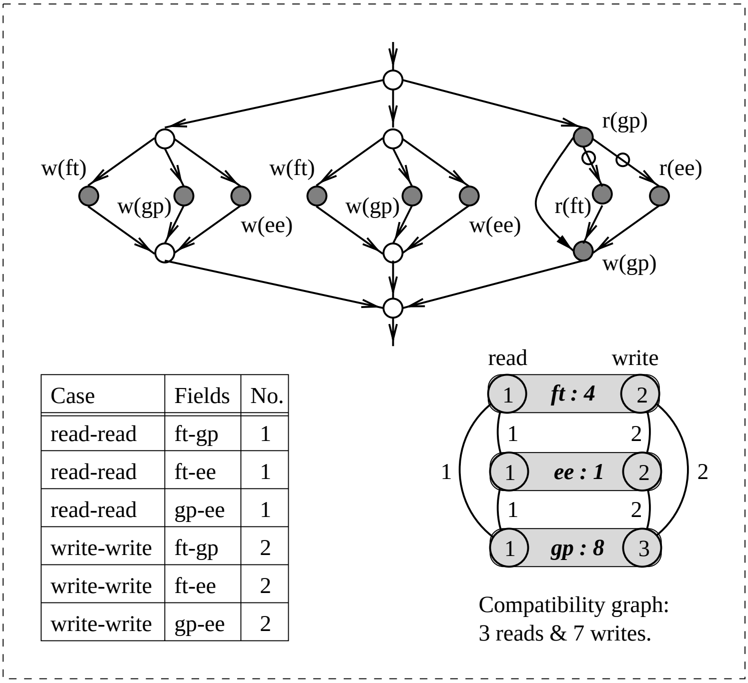
The first clustering step merged fields “ft” and “ee” thus reducing the number of memory accesses from 10 (3 reads and 7 writes) to 7 (2 reads and 5 writes). The new CDFG and compatibility graph are shown in Figure 7.13. The modified source code is shown in Figure 7.14.

The second clustering step miss-evaluated the weights in the compatibility graph and estimated that when merging all fields the resulting number of accesses would be 5 (2 reads and 3 writes). This was caused by the fact that the evaluation rules do not take into account how independent are parts of the CDFG. Having three separate compatibility graphs, like suggested in section 4.7., would have resulted in 4 new accesses (1 read and 3 writes). The resulting pruned CDFG and compatibility graph are shown in Figure 7.15. The modified source code is shown in Figure 7.16. This packing is used for all allocations in Table 7.1.

Group IHPL has only one basic group and therefore it was not packed in the first two pre-packing cases. An intuitive cross-module analysis of accesses, a future extension for the pre-packing methodology, allowed to merge this group with the group TBL1 (“pre-packing #3” in Table 7.1). This merging resulted in the reduction of both memory area and power consumption.

```
typedef struct {
  unsigned ft: 4;               // OAM function type
  unsigned gp: 8;               // Cell generation period
  unsigned ee: 1;               // End-to-end cell
} FMcellGener_record;

/*****  New command from Input Handler  *****/
  p1->ft = ft_in;                               // w(ft)
  p1->gp = FirstPeriod(ft_in);                  // w(gp)
  p1->ee = IsEndToEnd(ft_in);                   // w(ee)

/*****  New command from Physical Layer  *****/
  p2->ft = ft_cmd;                              // w(ft)
  p2->gp = FirstPeriod(ft_cmd);                 // w(gp)
  p2->ee = IsEndToEnd(ft_cmd);                  // w(ee)

/*****  At every 50 ms (signal from timer)  *****/
for ( ; ; ) {      // For every active Virtual Path
  tmp_gp = p3->gp;                              // r(gp)
  if ( tmp_gp == 1 ) {
    tmp_ft  = p3->ft;                           // r(ft)
    MakeAndSendCell( tmp_ft, p3->ee );          // r(ee)
    tmp_gp = RepeatPeriod(tmp_ft);
  else if ( tmp_gp > 1 )  tmp_gp--;
  p3->gp = tmp_gp;                              // w(gp)
}
```

Figure 7.11. Simplified source code of FMCG



Case	Fields	No.
read-read	ft-gp	1
read-read	ft-ee	1
read-read	gp-ee	1
write-write	ft-gp	2
write-write	ft-ee	2
write-write	gp-ee	2

Compatibility graph:
3 reads & 7 writes.

Figure 7.12. Pruned CDFG and compatibility graph of FMCG

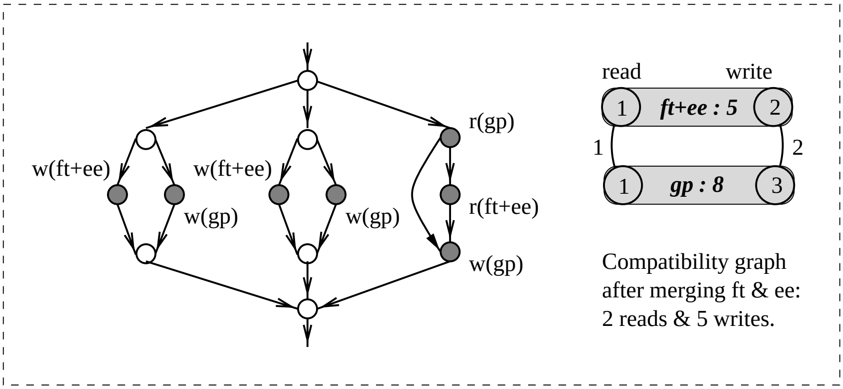


Figure 7.13. Clustering FMCG - step 1

```
typedef struct {
  unsigned ft_ee: 5;        // OAM function type & End-to-end cell
  unsigned gp: 8;           // Cell generation period
} FMcellGener_record;

/*****  New command from Input Handler  *****/
  p1->ft_ee = set_ft_ee( ft_in, IsEndToEnd(ft_in) );  // w(ft+ee)
  p1->gp = FirstPeriod(ft_in);                         // w(gp)

/*****  New command from Physical Layer  *****/
  p2->ft_ee = set_ft_ee( ft_cmd, IsEndToEnd(ft_cmd) ); // w(ft+ee)
  p2->gp    = FirstPeriod(ft_cmd);                     // w(gp)

/*****  At every 50 ms (signal from timer)  *****/
for ( ; ; ) {      // For every active Virtual Path
  tmp_gp = p3->gp;                                    // r(gp)
  if ( tmp_gp == 1 ) {
    tmp_ft_ee  = p3->ft_ee;                           // r(ft+ee)
    MakeAndSendCell( get_ft(tmp_ft_ee), get_ee(tmp_ft_ee) );
    tmp_gp = RepeatPeriod(get_ft(tmp_ft_ee));
  else if ( tmp_gp > 1 )  tmp_gp--;
  p3->gp = tmp_gp;                                    // w(gp)
}
```

Figure 7.14. Simplified source code of FMCG (after clustering step 1)

Group TBL1: The field “mode” is the result of intuitive pre-packing, applied when the F4/F5 block partitioned into synthesizable modules. Eight 1 bit fields - “AIS”, “RDI”, “loopback”, “monitor”, “activate”, “direction-1”, “direction-2” and “destination”, all indicating the mode of the ATM cell, were packed into the new field “mode”. A systematic analysis of dependency cases showed that there exist three different possibilities:

- all fields - “type”, “cmd” and “mode” - are kept as separate BGs, assigned to the same memory (size 1K*8), would result in 11 reads and 10 writes with power consumption 90.7;

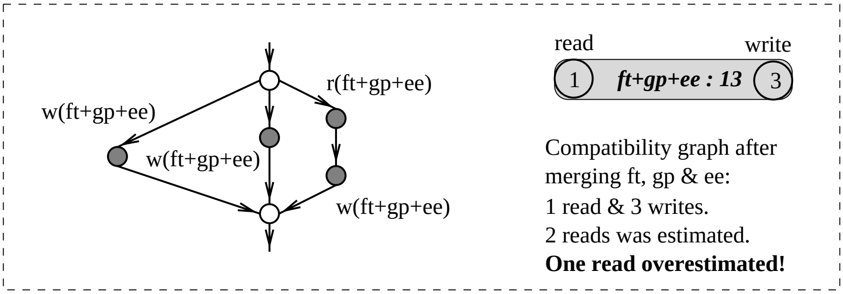


Figure 7.15. Clustering FMCG - step 2

```
typedef struct {
  unsigned ft_gp_ee: 13;   // OAM function type & Cell generation
                          // period & End-to-end cel
} FMcellGener_record;

/*****  New command from Input Handler  *****/
  p1->ft_gp_ee =                                     // w(ft+gp+ee)
    set_ft_ee( ft_in, FirstPeriod(ft_in), IsEndToEnd(ft_in) );

/*****  New command from Physical Layer  *****/
  p2->ft_gp_ee =                                     // w(ft+gp+ee)
    set_ft_ee( ft_cmd, FirstPeriod(ft_cmd), IsEndToEnd(ft_cmd) );

/*****  At every 50 ms (signal from timer)  *****/
for ( ; ; ) {      // For every active Virtual Path
  tmp_ft_gp_ee = p3->gp;                             // r(ft+gp+ee)
  tmp_gp       = get_gp(tmp_ft_gp_ee);
  if ( tmp_gp == 1 ) {
    MakeAndSendCell( get_ft(tmp_ft_gp_ee), get_ee(tmp_ft_gp_ee) );
    tmp_gp = RepeatPeriod(get_ft(tmp_ft_gp_ee));
  else if ( tmp_gp > 1 )  tmp_gp--;
  p3->ft_gp_ee = set_ft_ee( get_ft(tmp_ft_gp_ee),   // w(ft+gp+ee)
                            tmp_gp, get_ee(tmp_ft_gp_ee) );
}
```

Figure 7.16. Simplified source code of FMCG (after clustering step 2)

- fields “mode” and “cmd” packed into the same word, “type” kept separately and both resulting BGs assigned to the same single memory (size 512*16), would result in 8 reads and 6 writes with power consumption 67.2;
- all fields packed into the same word and assigned to a memory with size 256*32, would result in 5 reads and 5 writes with power consumption 60.0.

The third case demonstrates that even when a wider memory is used, with higher power consumption, of course, the smaller number of memory accesses reduces the total power consumption. The third packing is used for all allocations in Table 7.1.

It should be noted that different memory characteristics, e.g. different memory technologies, might give different values for area and power thus pointing onto different pre-packings to be

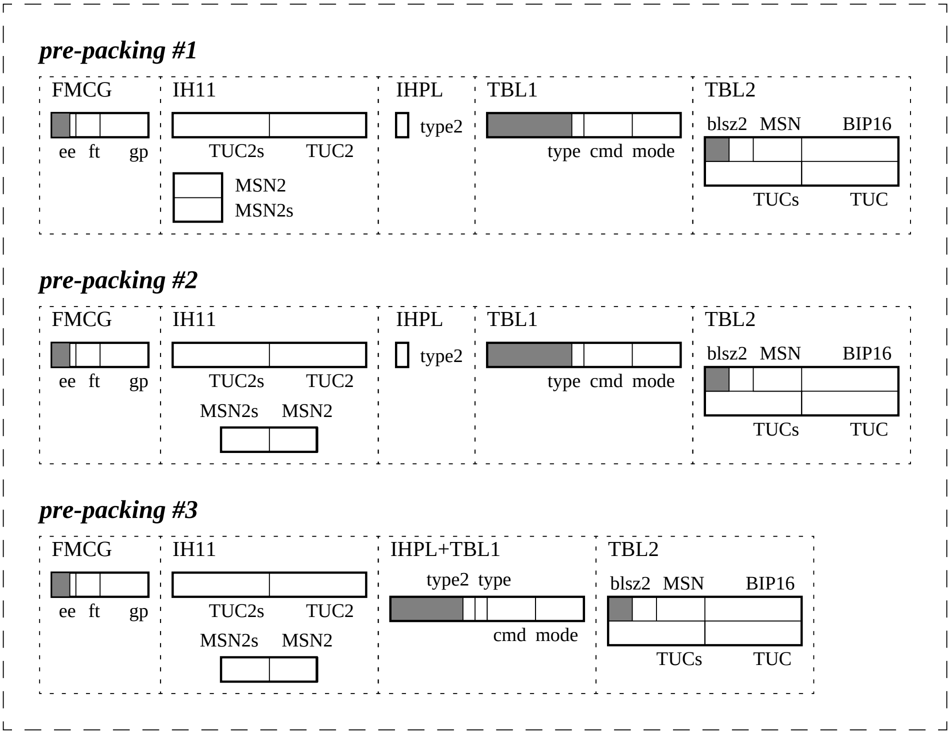


Figure 7.17. Memory allocations after systematic pre-packings

selected. Different pre-packings may result in different allocations, of course.

Group TBL2 was another simple case - all accesses are parallel. The use of 32-bit wide memories results in packing fields “TUC” and “TUCs”, both 16 bits, into a single word. The remaining three fields can be packed into a single word (without penalties for field “blsz2”).

Conclusion: Three different pre-packings were explored. Detailed results about the number of accesses and physical memory characteristics after allocation are presented in Table 7.1. All three cases reduced the total number of memory accesses from 76 to less than 40. At the same time the wasted memory area was reduced - the total area of memories was reduced by more than 3%. The most important result is that the total power consumption of memory accesses was reduced by more than 40%. All three cases, allocations shown in Figure 7.17., differ only in the packing and allocation of some of the groups. “Pre-packing #1” and “Pre-packing #2” differ in the allocation of group IH11, while in the case “Pre-packing #3” the group IPHL has been merged with group TBL1. Figure 7.18. compares the four different pre-packing cases in a more illustrative way.

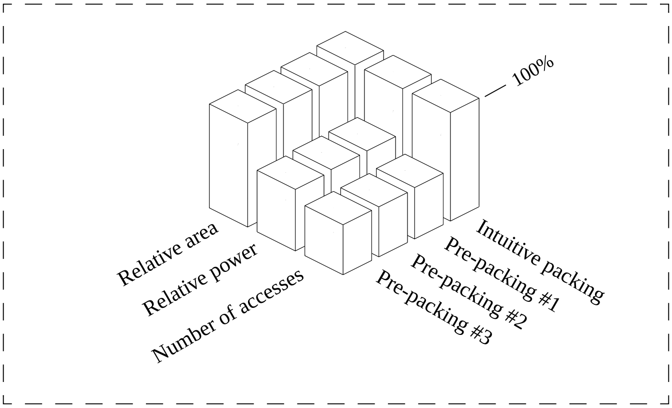


Figure 7.18. Comparison of different pre-packing cases

7.3. xTractor - synthesis results

xTractor - the HLS prototype tools set for CMIST applications, presented in chapter 2 - was used to synthesize the modules of the F4/F5 block. The initial partitioning, intuitive packing and all three pre-packing cases were used. Additionally two commercial HLS tools were used for comparison. The earlier CMIST synthesis approach (SYNT-X [HSE95], [SEP96]) was also used for comparison. Detailed synthesis results of modules are presented in Table 7.2. The case “Pre-packing #2” is omitted since it differs very little from the case “Pre-packing #1” - only module “Input Handler 1.1” is different - area 4435 gates instead of 4366. In all cases, the same backend synthesis tool was used with the same design constraints. The target technology was AMS 0.8 micron CMOS in 3.3 V.

The last case, “BG compaction”, represents a more significant change in the architecture. A methodology, called basic group compaction ([VMB99]), was experimentally applied to the arrays where indexing is known at compile time. These arrays are used for temporary storage of ATM cells in the modules of the F4/F5 block. Since bytes of a cell are always accessed in the same order, from the beginning of the array towards the end of the array, the indexes are always known. In the case “BG compaction”, cells are represented as arrays of 16-bit wide words. This resulted also with 16-bit wide buses between modules of the F4/F5 block, instead of the previous 8-bit buses. The total area of the design increased, of course, because of the wider data buses but the number of memory accesses was reduced again. Any module needs now 27 accesses instead of 53 to process an ATM cell. This results in the power reduction[1] of approximately 30% at the expense of area increase by 2%. Figure 7.19. compares synthesis results from different HLS tools of different modules of the F4/F5 block.

[1] Available memory models do not cover so small memories and the used numbers were approximated.

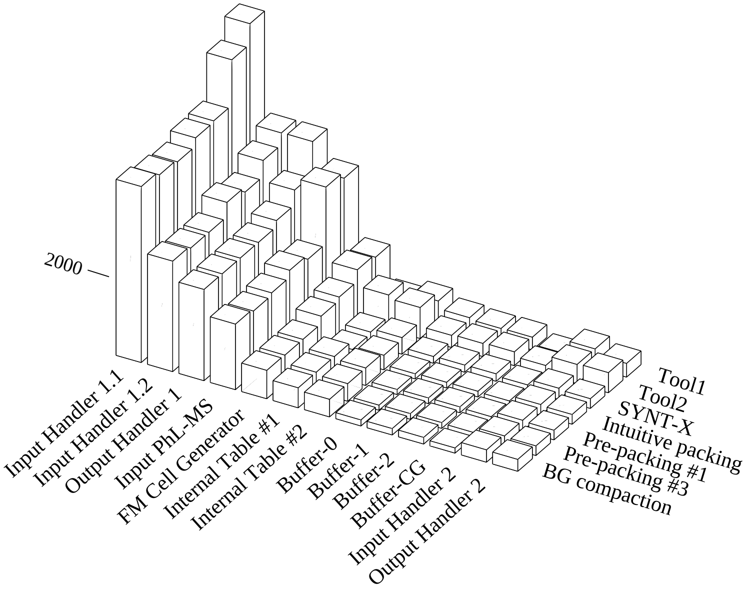


Figure 7.19. Comparison of synthesis results of OAM of ATM switch

Another series of experiments was made using some industrial designs. The synthesis results are shown in Figure 7.20. The results for comparison were obtained with some commercial synthesis tools as well as with manual designs, wherever available. A brief description of the design examples used for this purpose is given as follows:

- *Manchester encoder*: A small example written in two different ways - *mask* and *bit*.
- *Bus-arbitrator*: A small device to control the access to a shared bus.
- *Bofors*: A part of a control system in one of Bofor's products - synchronization of a slave clock with a master clock. *Measure* uses *calc* as sub-module.
- *Ericsson*: A sub-block in a GSM base station from Ericsson. The module performs correlation of GSM burst and calculates channel estimates.
- *F4/F5*: This is the same design used for detailed description of results. Here it is shown for comparisons.

Heights of the bars represent the areas of the synthesized designs in gates on a relative scale, normalized to the CMIST results.

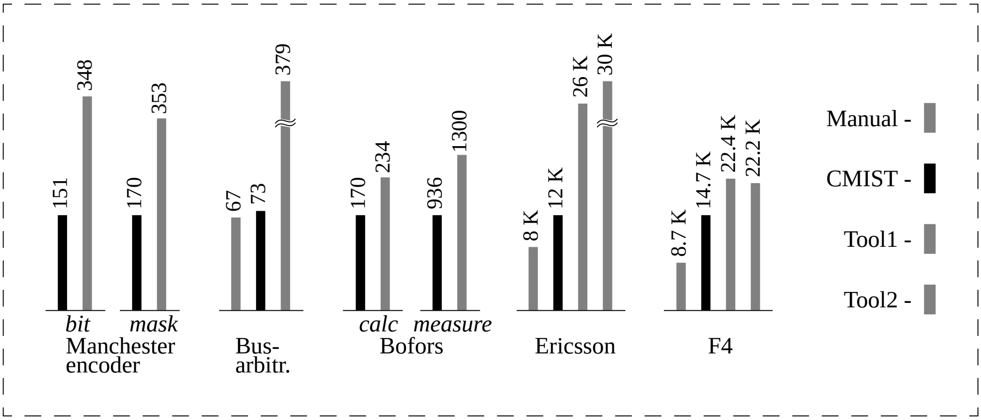


Figure 7.20. Synthesis results of various industrial design examples

Table 7.1. Data field pre-packing in OAM (F4/F5) of ATM switch

Group	Field	bits	Initial	Intuitive packing			Pre-packing #1			Pre-packing #2			Pre-packing #3		
			r / w	**r / w**	**area**	**power**	**r / w**	**area**	**power**	**r / w**	**area**	**power**	**r / w**	**area**	**power**
FMCG	m.ft	4	4 / 1	3 / 2	2.40%	3.19%	1 / 3	9.58%	5.11%	1 / 3	9.58%	5.11%	1 / 3	9.58%	5.11%
	m.gp	8	3 / 7	3 / 4	4.79%	6.71%		[256*16]			[256*16]			[256*16]	
	m.ee	1	1 / 3	1 / 2	0.60%	0.96%									
IH11	s.TUC2	16	5 / 2	5 / 2	9.58%	8.94%	2 / 2	19.16%	7.67%	2 / 2	19.16%	7.67%	2 / 2	19.16%	7.67%
	s.TUC2s	16	2 / 2	1 / 2	9.58%	3.83%		[256*32]			[256*32]			[256*32]	
	s.MSN2	8	3 / 2	2 / 2	4.79%	3.83%	2 / 2	9.58%	6.90%	3 / 2	9.58%	6.39%	3 / 2	9.58%	6.39%
	s.MSN2s	8	1 / 1	1 / 1	4.79%	1.92%	1 / 1	[512*8]			[256*16]			[256*16]	
IHPL	s.type2	2	2 / 2	2 / 2	1.20%	1.92%	2 / 2	1.20%	1.92%	2 / 2	1.20%	1.92%	5 / 5	19.16%	19.16%
TBL1	s.type	2	23 / 1	3 / 1	14.37%	28.97%	5 / 5	19.16%	19.16%	5 / 5	19.16%	19.16%		[256*32]	
	s.cmd	8	13 / 6	3 / 4	[1K*8]			[256*32]			[256*32]				
	s.mode	8	59 / 32	5 / 5											
TBL2	s.TUC	16	15 / 3	2 / 2	47.90%	39.73%	2 / 2	38.32%	18.40%	2 / 2	38.32%	18.40%	2 / 2	38.32%	18.40%
	s.TUCs	16	2 / 4	2 / 2	[2K*16]			[512*32]			[512*32]			[512*32]	
	s.BIP16	16	4 / 3	2 / 2			2 / 2			2 / 2			2 / 2		
	s.MSN	8	5 / 3	2 / 2											
	s.blsz2	4	1 / 3	1 / 1											
	Total		141/78	40/36	100%	100%	17/19	97.0%	59.2%	17/18	97.0%	58.6%	17/18	95.8%	56.7%

Table 7.2. Synthesis results of OAM (F4/F5) of ATM switch

Module	#	Tool 1	Tool 2	SYNT-X style (intuitive packing)		xTractor (intuitive packing)		xTractor (pre-packing #1)		xTractor (pre-packing #3)		xTractor (BG compaction)	
		area [area]	area [gates]	FSM states	area [gates]	FSM states	area [gates]	FSM states	area [gates]	FSM states	area [gates]	FSM states	area [gates]
Internal Table #1	1	384	887	23	335	13	291	5	514	5	514	5	514
Internal Table #2	1	563	796	24	425	13	373	5	451	5	451	5	451
Buffer-0	1	365	311	12	172	8	129	8	129	8	129	8	154
Buffer-1	2	320	340	12	184	8	144	8	144	8	144	8	164
Buffer-2	4	330	355	12	191	8	147	8	147	8	147	8	170
Buffer-CG	2	145	235	6	140	5	132	5	132	5	132	5	132
FM Cell Generator	1	1222	1303	58	1005	35	917	24	722	24	722	24	732
Input Handler 1.1	1	6310	5783	176	4614	158	4579	112	4366	112	4435	101	4538
Input Handler 1.2	1	3728	3423	158	3010	154	3150	105	2781	105	2781	94	2846
Input PhL-MS	1	3027	3203	87	1745	85	1866	63	1688	62	1624	55	1691
Input Handler 2	1	494	498	10	288	9	273	9	273	9	273	9	317
Output Handler 1	1	3740	2891	68	2510	28	2351	28	2351	28	2351	28	2350
Output Handler 2	1	351	519	14	252	9	248	9	248	9	248	9	305
Total		22,434	22,184	15,768		15,317		14,663		14,668		15,170	

8. Thesis Summary

Some aspects of High-Level Synthesis (HLS) methodology, targeting Control and Memory Intensive (CMIST) applications have been studied and analyzed in this thesis. In chapter 2, a prototype HLS tool set designed to test the methods developed in the scope of the thesis is presented. The internal representation (IRSYD), designed keeping in mind CMIST applications and used by the prototype tool is described in chapter 3. In chapter 4, one phase of high-level memory synthesis, namely the methodology for pre-packing of data fields of dynamically allocated data structures is described. The segment-based scheduling algorithm, targeting control flow dominated applications is presented in chapter 5. In chapter 6, fast heuristics for unified allocation and binding of functional units and storage elements are described. HLS results of industrially relevant design examples are discussed in chapter 7.

This chapter summarizes the conclusions of the sub-tasks presented in the thesis and outlines some suggestions to improve the presented prototype tool and methodologies.

8.1. Conclusions

Recent developments in microelectronic technology and CAD technology allow to produce larger and larger integrated circuits in shorter and shorter design times. At the same time, the abstraction level of specifications is getting higher both to cover the increased complexity of systems more efficiently and to make the design process less error prone. High-Level Synthesis (HLS) is one of the methodologies to handle the increased complexity. Although HLS has been successfully used in many cases, it is still not as indispensable today as layout or logic synthesis. Various application specific synthesis strategies have been developed to cope with the following problems ([GDW93], [HSE95], [Lin97], [BRN97]):

- there exist a need for application specific synthesis strategies which more efficiently could cooperate with the features of a specific application domain;
- existing internal representations can not encapsulate details of a specification without some loss of information, therefore more general representations are needed;
- the need for efficient estimation algorithms, especially in the deep sub-micron area where wiring dominates both gate delay and chip area;

Development of domain specific HLS tools has, though, a drawback. The more domain-specific a tool is, the smaller its customer base will be. A possible solution is to embed domain specific knowledge from wider areas. [Lin97]

In this thesis solutions for some of the sub-tasks of HLS, targeting control and memory intensive applications are presented. The sub-tasks cover the following topics:

- control flow oriented internal representation;
- methodology for memory mapping optimization by exploiting locality of data transfers;
- efficient control-flow based scheduling algorithm;
- fast heuristics for unified allocation and binding.

Prototype HLS tool set: xTractor, presented in chapter 2., is a prototype tool set developed to test High-Level Synthesis methodology of CMIST style applications. The tool set consists of several smaller component programs to solve certain synthesis steps, and of an interactive shell around the component programs. The component tools work on the internal representation IRSYD. The tool set generates RT-Level VHDL or Verilog code for Logic Level synthesis tools. Different target architectures can be selected for better exploitation of back-end tools.

IRSYD - Internal Representation for System Description: An intermediate design representation or an Internal Representation (IR) is always an important element of design methodologies and synthesis tools. It is many times designed just keeping in mind the convenience of translation and requirements of one tool (generally synthesizer or simulator). An IR is also important to represent specifications written in various high level languages like VHDL, C/C++, etc. Development of complex digital systems requires not only integration of various types of specifications, but also integration of different tools for analysis and synthesis. IRSYD, described in chapter 3., was developed to meet various requirements for internal representations. It is specifically targeted towards representation of heterogeneous systems with multiple front-end languages and to facilitate the integration of several design and validation tools.

Pre-packing of data fields in memory synthesis: The memory access bandwidth is one of the main design bottlenecks in control and data-transfer dominated applications such as protocol processing and multimedia. Memory allocation methodologies, which target static data structures (see e.g. [VCV89], [VLV95]), do not work well with dynamically allocated data structures (DADS). DADS are defined on the basis of functional and logical coherence for readability. Retaining such coherence while organizing the DADS physically in the memory does not optimize the required storage, nor does it optimize the required bandwidth.

These optimization goals are addressed by analysis of the dependencies between accesses and using them as the basis for packing elements of DADS into a single memory word. This pre-packing, described in chapter 4., minimizes not only the total number of data-transfers by merging memory accesses, but also minimizes the total storage by reducing bit-width wastage. The technique allows system designers to explore different memory data-layout alternatives at the system level when the application is still being defined.

Segment-based scheduling: Scheduling algorithms can be classified into two major categories: data-flow based and control-flow based. Data-flow based scheduling algorithms allow to use flexible cost functions and exploit parallelism in data dependencies. What they lack is the chaining of operations and treatment of control operations. The control-flow based schedulers allow, in principle, operator chaining and target mostly as-fast-as-possible schedules. A typical disadvantage of control-flow based schedulers is their complexity - they try to optimize all possible paths and they handle loops as ordinary paths. [Lin97]

The segment-based scheduler, presented in chapter 5., is control-flow centric and is free of the typical drawbacks of control-flow based schedulers. The scheduling approach avoids construction of all control paths (or their tree representation). This is achieved by dividing control graph into segments during the graph traversal. Segmentation drastically reduces the number of possible paths thus speeding up the whole scheduling process because one segment is analyzed at a time only. The segmentation also allows to schedule loop bodies efficiently.

Unified allocation and binding: Allocation and binding of functional units, storage elements and interconnections are important steps of any High Level Synthesis (HLS) tool. Most of the tools implement these steps separately ([GDW93], [DMi94]). It has been long realized that a unified binding, though harder, can lead to a better solution. Fast heuristic algorithms, presented in chapter 6., for solving the allocation and binding problems in a unified manner were developed for the prototype HLS tool set. The algorithms solve graph coloring problem by working on a weighted conflict graph. The unification of binding sub-tasks is taken care by the weights. The weights and cost functions model actual hardware implementations therefore allowing reliable estimations.

HLS synthesis examples: Some industrially relevant design examples were synthesized by using the methodologies presented in the thesis. The effectiveness of the methodologies was validated by comparing the results against synthesis results of commercially available HLS tools and in some cases against manually optimized designs. Some points of interest are discussed in details chapter 7.

8.2. Future Work

Prototype HLS tool set: The tool set was designed to solve specific synthesis tasks only. Therefore, several expansions are planned to add to target wider application areas. The most important expansions are as follows:

- translators to map different high level languages, e.g. VHDL, SDL and C/C++, into used internal representation (IRSYD);
- data and control flow transformations to reduce the impact of encoding style;
- automatization of data field pre-packing methodology described in chapter 4., and related memory mapping methodologies;
- elements of data flow oriented scheduling techniques for more efficient data flow synthesis;
- enhanced estimations for unified allocation and binding, plus false loop detection and removal to simplify back-end synthesis tasks.

IRSYD - Internal Representation for System Description: Various supporting tools are needed for efficient use of IRSYD as a unifying internal representation. These tools, additionally to ones solving different partitioning and synthesis tasks, should cover simulation and visualization of the control-flow of a design represented by IRSYD.

Pre-packing of data fields in memory synthesis: The following issues have been identified and are subjects of the future research:

- Combining dependency cases from semi-independent parts of the pruned CDFG may have a cumulative effect - clustering may give different results depending on the actual CDFG (see Figures 4.12. and 4.13.). A step-by-step clustering is one possible solution - every decision of pre-packing is followed by the rebuilding of the pruned dependency graph. Another solution is to expand the compatibility graph to take into account the independence of the access groups.
- Usage of hyper-edges in the compatibility graph for better generalization. The hyper-edges allow also packing of more than two BG-s at a time. Converting edges into hyper-edges can be too time consuming, though.
- Better power and area estimation algorithms are needed for finer analysis of trade-offs.
- Long dependency chains, and cases “read-or-write” and “write-or-write”, not analyzed in the scope of the proposed approach, are also subjects of future research.

Segment-based scheduling: Various data flow based scheduling techniques and transformations can be used to improve control flow centric schedulers. The best candidates are techniques, which allow to move operations out of basic blocks, thus allowing speculative execution, for instance. Another benefit of allowing operations to be moved is creating incremental improvements of the overall schedule. Potential path explosions, caused by deep nested if-statements, can be avoided by converting control flow constructs into data flow constructs. All these transformations not only would help to improve generated schedules, but also would widen the application area.

Unified allocation and binding: Reuse of chained operations creates a possibility for asynchronous feedbacks, called false loops; e.g. two reused arithmetic units are chained into a single clock step in different order in different states. False loops can be caused also by sharing also in control path. This situation can be generated actually by any binding algorithm. Although the possibility for such a case is very low in CMIST style applications, and never occurred in practice, it should be taken into account anyway. A methodology, which detects and eliminates false loops both from data path and control path, is proposed in [SLH96].

The coloring algorithms can be speed-up by creating separate conflict graphs for different groups of functional units. This allows to avoid attempts to color nodes with incompatible type with the same color. The branch-and-bound based heuristic algorithm can be improved by reordering the search tree branches in such a way that the probability of hitting enough good solutions at earlier phases is increased.

9. References

[ASU86] A. V. Aho, R. Sethi, J. D. Ullman, “Compilers: Principles, Techniques, and Tools”, Addison-Wesley Publishing Company, 1986, ISBN 0-201-10194-7.

[BMB88] M. Balakrishnan, A. K. Majumdar, D. K. Banerji, J. G. Linders, J. C. Majithia, “Allocation of multiport memories in data path synthesis” IEEE Trans. on Comp.-aided Design, Vol.CAD-7, No.4, pp.536-540, Apr. 1988.

[Bal95] F. Balasa, “Background memory allocation for multi-dimensional signal processing”, Doctoral dissertation, ESAT/EE Dept., K.U.Leuven, Belgium, Nov. 1995.

[BaSk86] S. I. Baranov, V. A. Sklyarov, “LSI Digital Devices on Programmable Matrix Structures” (in Russian), Radio i svyaz, 1986.

[BDB94] S. Bhattacharya, S. Dey, F. Brglez, “Performance Analysis and Optimization of Schedules for Conditional and Loop-Intensive Specifications”, Proc. of 31st ACM/IEEE Design Automation Conference, pp. 491-496, June 1994.

[BRN97] R. Bergamaschi, S. Raje, I. Nair, L Trevillyan, “Control-Flow Versus Data-Flow Based Scheduling: Combining Both Approaches in and Adaptive Scheduling System”, IEEE Transactions on VLSI Systems, Vol. 5., No. 1., pp. 82-100, March 1997.

[BeMe84] R. E. Berry, B. A. E. Meekings, “A Book on C”, Macmillan Publishers, 1984, ISBN 0-333-36821-5.

[BDS95] K. Bilinski, E. Dagless, J. Saul, “Behavioral Synthesis of Complex Parallel Controllers”, Proc. of 9th Intnl. Conference on VLSI Design, Jan. 1996.

[CaBe90] R. Camposano, R. Bergamaschi, “Synthesis Using Path-Based Scheduling: Algorithms and Exercises”, Proc. of 27th ACM/IEEE Design Automation Conference, pp. 450-455, June 1990.

[Cam91] R. Camposano, “Path-Based Scheduling for Synthesis”, IEEE Transactions on Computer-Aided Design, Vol. 10, No. 1, pp. 85-93, January 1991.

[CGD94] F. Catthoor, W. Geurts, H. De Man, “Loop transformation methodology for fixed-rate video, image and telecom processing applications”, Proc. of Intnl. Conf. on Applic.-Spec. Array Processors, San Francisco, CA, pp. 427-438, Aug. 1994.

[CWD98] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, A. Vandecappelle, “Custom Memory Management Methodology -- Exploration of Memory Organisation for Embedded Multimedia System Design”, Kluwer Acad. Publ., Boston, 1998, ISBN 0-7923-8288-9.

[CGJ94] M. Chiodo, P. Giusto, A. Jurecska, A. Sangiovanni-Vincentelli, L. Lavagno, “Hardware-software codesign of embedded systems”, IEEE Micro vol. 14, no. 4, pp. 26-36, Aug. 1994.

[CMIST-www] CMIST. “http://www.ele.kth.se/~lrv/CMIST”.

[Cosima-www] The COSYMA system. Technische Universität Braunschweig.
“http://www.ida.ing.tu-bs.de/projects/cosyma/”.

[Cou96] O. Coudert, “On Solving Covering Problems”, Proc. of 33rd ACM/IEEE Design Automation Conf., Las Vegas, NV, USA, June 1997

[Cou97] O. Coudert, “Exact Coloring of Real-Life Graphs is Easy”, Proc. of 34th ACM/ IEEE Design Automation Conf., Anaheim, CA, USA, June 1997.

[DMi94] G. De Micheli, “Synthesis and Optimization of Digital Circuits”, McGraw-Hill, Inc., 1994, ISBN 0-07-113271-6.

[DePa96] T. Denk, K. Parhi, “Lower bounds on memory requirements for statically scheduled DSP programs”, Journal of VLSI Signal Processing, No.12, pp.247-263, 1996.

[DHE94] A. Doboli, J. Hallberg, P. Eles, “A Simulation Model for the Operation and Maintenance Functionality in ATM Switches”, Internal report, Dept. of Computer and Information Science, University of Linköping, Sweden, 1994.

[DHG90] N. Dutt, T. Hadley, and D. Gajski, “An Intermediate Representation for Behavioural Synthesis”, Proc. of 27th ACM/IEEE Design Automation Conference, pp. 14-19, June 1990.

[EKP98] P. P. Eles, K. Kuchcinski, Z. Peng, “System Synthesis with VHDL”, Kluwer Academic Publisher, 1998, ISBN 0-7923-8082-7.

[EHK96] P. Ellervee, A. Hemani, A. Kumar, B. Svantesson, J. Öberg, H. Tenhunen, “Controller Synthesis in Control and Memory Centric High Level Synthesis System”, The 5th Biennial Baltic Electronic Conference, Tallinn, Estonia, October 1996.

[EKS96] P. Ellervee, A. Kumar, B. Svantesson, A. Hemani, “Internal Representation and Behavioral Synthesis of Control Dominated Applications”, The 14th NORCHIP Conference, Helsinki, Finland. Nov. 1996.

[EKS96b] P. Ellervee, A. Kumar, B. Svantesson, A. Hemani, “Segment-Based Scheduling of

Control Dominated Applications in High Level Synthesis", International Workshop on Logic and Architecture Synthesis, Grenoble, France, Dec. 1996.

[EKH97] P. Ellervee, S. Kumar, A. Hemani, "Comparison of Four Heuristic Algorithms for Unified Allocation and Binding in High-Level Synthesis", The 15th NORCHIP Conference, Tallinn, Estonia, pp. 60-66, Nov. 1997.

[EKJ97] P. Ellervee, S. Kumar, A. Jantsch, A. Hemani, B. Svantesson, J. Öberg, I. Sander, "IRSYD - An Internal representation for System Description (Version 0.1)", Internal Report TRITA-ESD-1997-10, 1997 KTH ELE ESDLab, Stockholm, Sweden.

[EKJ98] P. Ellervee, S. Kumar, A. Jantsch, B. Svantesson, T. Meincke, A. Hemani, "IRSYD: An Internal Representation for Heterogeneous Embedded Systems", The 16th NORCHIP Conference, pp.214-221, Lund, Sweden, Nov. 1998.

[EMC99] P. Ellervee, M. Miranda, F. Catthoor, A. Hemani, "Exploiting Data Transfer Locality in Memory Mapping", Proc. of 25th Euromicro Conference, pp.14-21, Milan, Italy, Sept. 1999.

[EMC00] P. Ellervee, M. Miranda, F. Catthoor, A. Hemani, "High-level Memory Mapping Exploration for Dynamically Allocated Data Structures". Submitted to the 37th ACM/IEEE Design Automation Conference, Los Angeles, CA, USA, June 2000.

[FRJ94] Y. Fann, M. Rim, R. Jain, "Global Scheduling for High-Level Synthesis Applications", Proc. of 31st ACM/IEEE Design Automation Conference, June 1994.

[FeHi93] N.Fenton, G.Hill, "Systems Construction and Analysis: A Mathematical and Logical Framework", McGraw-Hill International Limited, 1993, ISBN 0-07-707431-9.

[GaKu83] D. Gajski, R. Kuhn, "Guest Editor's Introduction: New VLSI tools", IEEE Computer, Vol.16, No.12, pp.11-14, Dec. 1983.

[GDW93] D. Gajski, N. Dutt, A. Wu, S. Lin, "High-Level Synthesis: Introduction to Chip and System Design", Kluwer Academic Publishing, 1993.

[GVN94] D. Gajski, F. Vahid, S. Narayan, J. Gong, "Specification and Design of Embedded Systems", Prentice Hall, 1994.

[GRV90] G. Goossens, J. Rabaey, J. Vandewalle, H. De Man, "An efficient microcode compiler for application-specific DSP processors", IEEE Trans. on Comp.-aided Design, Vol.9, No.9, pp.925-937, Sep. 1990.

[Hem92] A. Hemani, "High-Level Synthesis of Synchronous Digital Systems using Self-Organisation Algorithms for Scheduling and Binding", Ph.D. Thesis ISRN KTH/TDE/FR--92/5-

-SE, Stockholm, 1992.

[HSE95] A. Hemani, B. Svantesson, P. Ellervee, A. Postula, J. Öberg, A. Jantsch, H. Tenhunen, “High-Level Synthesis of Control and Memory Intensive Communications System”, Eighth Annual IEEE International ASIC Conference and Exhibit (ASIC’95), pp.185-191, Austin, USA, Sept. 1995.

[HLH93] S. H. Huang, Y. L. Jeang, C. T. Hwang, Y. C. Hsu, J. F. Wang, “A Tree-Based Scheduling Algorithm for Control-Dominated Circuits”, Proc. of 30th ACM/IEEE Design Automation Conference, pp. 578-582, June 1993.

[IsJe95] T. Ben Ismail, A.A. Jerraya, “Synthesis Steps and Design Models for Codesign”, IEEE Computer, pp. 44 - 52, Feb. 1995.

[JeOB95] A.A. Jerraya, K. O’Brien, “SOLAR: An Intermediate Format for System-Level Modeling and Synthesis”, Codesign: Computer-aided Software/Hardware Engineering, IEEE Press, Jerzy Rozenblit and Klaus Buchenrieder, 7, pp. 145-175, 1995.

[KiPo98] D. Kirovski, M. Potkonjak, “Efficient Coloring of a Large Spectrum of Graphs”, Proc. of ACM/IEEE 35th Design Automation Conference, pp. 427-432, San Francisco, USA, June 1998.

[KuBh79] A. Kumar, P.C.P. Bhatt, “A structured Language for Digital System Design”, Proc. of International Symposium on Computer Architecture, La Boule, France, 1979.

[KSH97] S. Kumar, B. Svantesson, A. Hemani, “A Methodology and Algorithms for efficient synthesis from System Description in SDL”, Technical Report, No TRITA-ESD-1997-06, ESDlab, Royal Institute of Technology, Stockholm, Sweden, 1997.

[KuCh97] M.-T. Kuo, C.-K. Cheng, “A Network Flow Approach for Hierarchical Tree Partitioning”, Proc. of 34th ACM/IEEE Design Automation Conf., Anaheim, CA, June 1997.

[KuPa87] F. J. Kurdahi, A. C. Parker, “REAL: a program for register allocation”, Proc. of 24th ACM/IEEE Design Automation Conf., Miami FL, pp.210-215, June 1987.

[KuAb95] P. Kurup, T. Abbasi, “Logic Synthesis Using Synopsys”, Kluwer Academic Publisher, 1995, ISBN 0-7923-9582-4.

[LaTh91] E.D. Lagnese, D.E. Thomas, “Architectural Partitioning for System Level Synthesis of Integrated Circuits”, IEEE Trans. on CAD, Vol.10, No.7, 1991.

[LRJ98] G. Lakshminarayana, A. Raghunathan, N. Jha, “Incorporating Speculative Execution into Scheduling of Control-flow Intensive Behavioral Descriptions”, Proc. of 35th ACM/IEEE Design Automation Conf., San Francisco CA, pp.108-113, June 1998.

[LKK92] M. Lauria, S. Kumar, A. Kumar, “A partitioning scheme for multiple PLA based Control part Synthesis”, Proc. of Intnl. Conference on VLSI Design, Bangalore, 1992.

[Lee76] S. C. Lee, “Digital Circuits and Logic Design”, Prentice-Hall, Inc., 1976.

[LiHe98] Y. Li, J. Henkel, “A framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems”, Proc. of 35th ACM/IEEE Design Automation Conf., pp. 188-193, June 1998.

[Lin97] Y.-L. Lin, “Recent Developments in High-Level Synthesis”, ACM Transactions on Design Automation of Electronic Systems, Vol. 2, No.1, pp. 2-21, Jan. 1997.

[McKo90] M.C. McFarland, T.J. Kowalski, “Incorporating Bottom-Up Design into Hardware Synthesis”, IEEE Trans. on CAD, Vol.9, No.9, 1990.

[MJE99] T. Meincke, A. Jantsch, P. Ellervee, A. Hemani, H. Tenhunen, “A Generic Scheme for Communication Representation and Mapping”, The 17th NORCHIP Conference (poster), pp.334-339, Oslo, Norway, Nov. 1999.

[MLD92] P. Michel, U. Lauther, P. Duzy, editors, “The Synthesis Approach to Digital System Design”, Kluwer Academic Publisher, 1992, ISBN 0-7923-9199-3.

[MCJ96] M. Miranda, F. Catthoor, M. Janssen, H. De Man, “Adopt: Efficient hardware address generation in distributed memory architectures”, Proc. of 9th ACM/IEEE Intnl. Symp. on System-Level Synthesis, La Jolla CA, USA, pp. 20-25, Nov. 1996.

[MCJ98] M. Miranda, F. Catthoor, M. Janssen, H. De Man, “High-Level Address Optimization and Synthesis Techniques for Data-Transfer-Intensive Applications”, IEEE Trans. on VLSI Syst., Vol.6, No.4, Dec. 1998.

[Moo96] G. Moore, “Nanometers and Gigabucks - Moore on Moore’s Law”, University Video Corporation Distinguished Lecture, 1996, (http://www.uvc.com).

[Obe96] J. Öberg, “An Adaptable Environment for Improved High-Level Synthesis”, Licentiate Thesis ISRN KTH/ESD/R--96/14--SE, 1996.

[Obe99] J. Öberg, “ProGram: A Grammar-Based Method for Specification and Hardware Synthesis of Communication Protocols”, Ph.D. Thesis ISRN KTH/ESD/AVH--99/3--SE, Stockholm, 1999.

[ONi96] M. O’Nils, “Hardware/Software Partitioning of Embedded Computer Systems”, Licentiate Thesis ISRN KTH/ESD/R--96/17, Stockholm, 1996.

[ONi97] M. O’Nils, “Communication within HW/SW Embedded Systems”, ESDLab, Department of Electronics, Royal Institute of Technology, Sweden, report no. TRITA-ESD-1997-08, ESDlab, KTH-Electrum, Electrum 229, S-16440 Kista, Sweden, 1997.

[ONi99] M. O'Nils, "Specification, Synthesis and Validation of Hardware/Software Interfaces", Ph.D. Thesis ISRN KTH/ESD/AVH--99/4--SE, Stockholm, 1999

[OlTi98] K. Olcoz, J.F. Tirado, "Register Allocation with Simultaneous BIST Intrusion", 24th EUROMICRO Conference, pp. 99-106, Västerås, Sweden, August 1998.

[Ous94] J. K. Ousterhout, "Tcl and the Tk Toolkit", Addison-Wesley Publishing Company, 1994, ISBN 0-201-63337-X.

[PRS98] R. Passerone, J. A. Rowson, A. Sangiovanni-Vincentelli, "Automatic Synthesis of Interfaces between Incompatible Protocols", Proc. of ACM/IEEE 35th Design Automation Conference, pp. 8-13, San Francisco, USA, June 1998.

[Pau88] P. Paulin, "High-Level Synthesis of Digital Circuits Using Global Scheduling and Binding Algorithms", Ph.D. Thesis, Carleton, 1988.

[Pen87] Z. Peng, "A Formal Methodology for Automated Synthesis of VLSI Systems", Ph.D. Dissertation, No. 170, Dept. of Computer and Information Science, Linköping University, 1987.

[RaBr96] I. Radivojević, F. Brewer, "A New Symbolic Technique for Control Dependent Scheduling", IEEE Trans. on CAD of IC and Systems, Vol. 15, No. 1, Jan. 1996.

[ROJ94] M. Rahmouni, K. O'Brien, A. Jerraya, "A Loop-Based Scheduling Algorithm for Hardware Description Languages", Journal of Parallel Processing Letters, Vol. 4, No. 3, pp. 351-364, 1994.

[RaJe95] M. Rahmouni, A. Jerraya, "PPS: A Pipeline Path-based Scheduler", Proc. of 6th ACM/IEEE Europ. Design and Test Conf., Paris, France, Feb. 1995.

[RaJe95b] M. Rahmouni, A. Jerraya, "Formulation and Evaluation of Scheduling Techniques for Control Flow Graphs", Proc. of EURO-DAC'95, 1995.

[RGC94] L. Ramachandran, D. Gajski, V. Chaiyakul, "An algorithm for array variable clustering", Proc. of 5th ACM/IEEE Europ. Design and Test Conf., Paris, France, pp.262-266, Feb. 1994.

[Rai98] P. Raines, "Tcl/Tk: Pocket Reference", O'Reilly & Associates, Inc., 1998, ISBN 1-56592-498-3.

[SaBr95] G. Satir, D. Brown, "C++: The Core Language", O'Reilly & Associates, Inc., 1995, ISBN 1-56592-116-X.

[ScTh97] H. Schmit, D. Thomas. "Synthesis of Application-Specific Memory Designs", IEEE Trans. on VLSI Systems, Vol.5, No.1, pp.101-111, Mar. 1997.

[SSP92] M. Schönfeld, M. Schwiegershausen, P. Pirsch, "Synthesis of intermediate memories for the data supply to processor arrays," in Algorithms and Parallel Architectures II, P. Quinton, Y. Robert (eds.), Elsevier, Amsterdam, 1992, pp.365-370.

[ShCh99] W.-T. Shiue, C. Chakrabarti, "Memory Exploration for Low Power, Embedded Systems", Proc. of 36th ACM/IEEE Design Automation Conf., pp. 140-145, June 1999.

[SYM98] J. L. da Silva Jr., C. Ykman-Couvreur, M. Miranda, K. Croes, S. Wuytack, G. de Jong, F. Catthoor, D. Verkest, P. Six, H. De Man, "Efficient System Exploration and Synthesis of Applications with Dynamic Data Storage and Intensive Data Transfer", Proc. of 35th ACM/IEEE Design Automation Conf., San Francisco CA, pp.76-81, June 1998.

[SWC97] P. Slock, S. Wuytack, F. Catthoor, G. de Jong, "Fast and extensive system-level memory exploration for ATM applications", Proc. of 10th ACM/IEEE Intnl. Symp. on System-Level Synthesis, Antwerp, Belgium, pp.74-81, Sep. 1997.

[SSM93] E. Sternheim, R. Singh, R. Madhavan, Y. Trivedi, "Digital Design and Synthesis with Verilog HDL", Automata Publishing Company, 1993, ISBN 0-9627488-2-X.

[SLH96] A. Su, T.-Y. Liu, Y.-C. Hsu, M. Lee, "Eliminating False Loops caused by Sharing in Control Path", Proc. 9th ACM/IEEE Intnl. Symp. on System-Level Synthesis, La Jolla CA, USA, pp. 39-44, Nov. 1996

[SEP96] B. Svantesson, P. Ellervee, A. Postula, J. Öberg, A. Hemani, "A Novel Allocation Strategy for Control and Memory Intensive Telecommunication Circuits", The 9th International Conference on VLSI Design, Bangalore, India, January 1996.

[Sva96] B. Svantesson, "Modeling of OAM for ATM in VHDL", Master Thesis, TRITA-ESD-1994-05, ISRN KTH/ESD/R--94/05--SE, Royal Institute of Technology, Stockholm, Sweden, Dec. 1996.

[SKE98] B. Svantesson, S. Kumar, P. Ellervee, A. Hemani, "The Translation of SDL Descriptions to the Internal Representation IRSYD", Internal Report TRITA-ESD-1998-8, 1998 KTH ELE ESDLab, Stockholm, Sweden.

[Syn92a] Command Reference Manual. Synopsys, V3.0, Dec. 1992.

[Syn92b] Design Analyzer Reference Manual. Synopsys, V3.0, Dec. 1992.

[Syn92c] dc_shell scripts for Synthesis. Application Note. Synopsys, V3.0, Feb. 1993.

[Syn92d] Finite State Machines. Application Note. Synopsys, V3.0, Feb. 1993.

[Syn92e] Flattening and Structuring: A Look at Optimization Strategies. Application Note. Synopsys, V3.0, Feb. 1993.

[TWL95] A. Takach, W. Wolf, M. Leeser, “An Automaton Model for Scheduling Constraints in Synchronous Machines”, IEEE Transactions on Computers, Vol.44, No.1, pp.1-12, IEEE, January 1995.

[Tan96] A. Tanenbaum, “Computer Networks”, 3-rd edition, Prentice Hall, 1996.

[TclTk-www] Tcl/Tk. “http://www.scriptics.com/products/tcltk/”.

[VLV95] J. Van Meerbergen, P. Lippens, W. Verhaegh, A. van der Werf, “PHIDEO: high-level synthesis for high throughput applications”, Journal of VLSI signal processing, special issue on “Design environments for DSP” (eds. I. Verbauwhede, J. Rabaey), Vol.9, No.1/2, Kluwer, Boston, pp.89-104, Jan. 1995.

[VMB99] A. Vandecappelle, M. Miranda, E. Brockmeyer, F. Catthoor, D. Verkest, “Global Multimedia System Design Exploration using Accurate Memory Organization Feedback”, Proc. of 36th ACM/IEEE Design Automation Conf., pp. 327-332, June 1999.

[VCV89] I. Verbauwhede, F. Catthoor, J. Vandewalle, H. De Man, “Background memory management for the synthesis of algebraic algorithms on multi-processor DSP chips”, Proc. of VLSI’89, Int. Conf. on VLSI, Munich, Germany, pp.209-218, Aug. 1989.

[VCV91] I. Verbauwhede, F. Catthoor, J. Vandewalle, H. De Man, “In-place memory management of algebraic algorithms on application-specific IC’s”, Journal of VLSI signal processing, Vol.3, Kluwer, Boston, pp.193-200, 1991.

[Wak94] J. F. Wakerly, “Digital Design: Principles & Practices”, Prentice-Hall, Inc., 1994.

[WBL94] N. Wehn, J. Biesenack, T. Langmaier, M. Münch, M. Pilsl, S. Rumler, P. Duzy, “Scheduling of Behavioural VHDL by Retiming Techniques”, Proc. of EURO-DAC’94, pp. 546-551, ACM/IEEE, September 1994.

[WCD96] S. Wuytack, F. Catthoor, G. De Jong, B. Lin, H. De Man, “Flow Graph Balancing for Minimizing the Required Memory Bandwidth”, Proc. 9th ACM/IEEE Intnl. Symp. on System-Level Synthesis, La Jolla CA, USA, pp. 127-132, Nov. 1996.

[WDC98] S. Wuytack, J.-P. Diguet, F. Catthoor, H. De Man, “Formalized Methodology for Data Reuse Exploration for Low-Power Hierarchical Memory Mappings”, IEEE Transactions on VLSI Systems, Vol. 6, No. 4, pp. 529-537, Dec. 1998.

[YeWo96] T.-Y. Yen, W. Wolf, “An Efficient Graph Algorithm for FSM Scheduling”, IEEE Transactions on VLSI Systems, Vol. 4, No. 1, pp. 98-112, March 1996.

[YuKu98] S.-Y. Yuan, S.-Y. Kuo, “A New Technique for Optimization Problems in Graph Theory”, IEEE Trans. on Computers, Vol. 47, No. 2, pp. 190-196, Feb. 1998.

Appendix A. Command line options of component tools

CDFG checker:

```
----- Checking XFC - L(R)V - ver. O.y (99-11-28) -----

usage: xfc_chk [<options>] XFC-file

Options:
  -m    - list all memories, their organization and size
  -s    - list all states and their location
  -c    - show the connection table
  -all  - list above mentioned information
  -t    - find the number of all paths (can be very slow!)
  -o <file>   - save the XFC (to debug parse & dump functions)
  -on <file>  - the same without port/signal buffering
```

Data-path Analyzer:

```
----- XFC data-path analysis - L(R)V - ver. O.y (99-11-..) -----

usage:  xfc_data [<options>] source-XFC [target-XFC]

No output is produced when target-XFC is omitted.

Options:
 Optimize (performed in the order listed below):
  -const    - propagate constants (simplified data flow analysis only)
  -replace  - replace operations with constants (‘*’ & ‘/’ only)

 Report:
  -arithm     - analyze and list arithmetic operations
  -a’<list>’  - comma separated list of arithmetic operations
                (default is all)
  -delay      - estimate largest delays of blocks (unmarked XFC)
  -delay      - estimate delays between states (marked XFC)
  -d’<state>’ - detailed delays in all paths staring from <state>
  -perf       - performance in clock-steps (marked XFC only)
                This can be very time and memory consuming!

 Settings:
  -short      - short reports only (detailed reports by default)
```

Memory Extractor:

```
----- XFC - extracting memories - L(R)V - ver. O.y (99-11-..) -----

usage: xfc_memxtr -report [<options>] XFC-file
       xfc_memxtr [<options>] source-XFC target-XFC

Options:
 -map <file>  - make/use memory mapping file (use “-report” for making)
 -extr <list> - space/comma separated list of memories to be extracted
                (all by default, masks the list given by “-map”)

Mapping commands (“-map <file>”):
  Map onto another array         “map[ping]  <array> <target> <base>”
  Keep as a register file        “reg[ister] <array>”
  Extract as an external memory  “ext[ernal] <array>”
    <array>, <target> - names of arrays
    <base> - base address (C-style decimal, hexadecimal or octal number;
             corresponds to the lowest address of the source memory)
```

State Marker:

```
----- Extracting FSM from XFC - L(R)V - ver. O.y (99-03-..) -----

usage: xfc2fsm [<delay>] [-M<marking>] [-R<report>] source-XFC [target-XFC]

Delay analysis:
  “-clock <period>” - designer defined clock period [ns],
             (max block delay is used by default)
  “-D[<loop>][,<path>]” - delay constraints:
    <loop> - max delay when collecting loops [clock period],
             (5 by default, 0 - unlimited)
    <path> - max delay when collecting paths [clock period],
             (2 by default, 0 - unlimited)

Marking strategy “-M...”:
  i - try to follow “clock true” model for input ports
  o - try to follow “clock true” model for output ports
  k - keep existing state markings
  a - state marking - next state as marked (slower)
  d - state marking - double checking for changes (slower)
  e - skip the entry marking (if possible, can be problematic!)
  E - forcing the entry marking (always the first marking)

Reporting “-R...”:
  a - interactive state marking
  h - the same with confirmation for “no changes”
  r - reasoning about the state markings
  t - full profiling statistics of the scheduling
  d - list blocks with the largest delays (>=90%, in ns),
      (no state marking, target-XFC is ignored)
```

State Marker (Synt-X style)

```
----- Extracting FSM from XFC (SYNT-X) - L(R)V - ver. O.y (99-03-19) -----

usage: xfc_syntx [-M<marking>] [-R<report>] source-XFC target-XFC

Marking strategy “-M...”:
  k - keep existing state markings
  e - skip the entry marking (if possible, can be problematic!)

Reporting “-R...”:
  r - reasoning about the state markings
```

Allocator / Binder

```
----- Binding data-path components - L(R)V - ver. O.y (99-11-..) -----

usage:  xfc_bind [<options>] source-XFC [target-XFC]

No output is produced when target-XFC is omitted.

Options:
 Optimize (performed in the order listed below):
  -bind       - reallocate and bind arithmetic operations
  -b’<list>’  - comma separated list of arithmetic operations
                (default is all)
  -reg        - reallocate and bind storage elements (registers)

 Report:
  -arithm     - analyze and list arithmetic operations
  -a’<list>’  - comma separated list of arithmetic operations
                (default is all)
  -delay      - estimate largest delays of blocks (unmarked XFC)
  -delay      - estimate delays between states (marked XFC)
  -d’<state>’ - detailed delays in all paths staring from <state>
  -perf       - performance in clock-steps (marked XFC only)
                This can be very time and memory consuming!

 Settings:
  -short      - short reports only (detailed reports by default)
  -table      - dump the conflict table(s) only (“painter” format)
  -rand/-dyn  - pseudo random or dynamic greedy algorithm
                (static by default)
  -seed <nr>  - initial seed for random generator (0 by default)
  -try <nr>   - number of optimization tries (automatic by default)
  -bndwd <nr> - width coefficient for binding (2 by default)
  -small      - starting from smaller components (static greedy)
  -select     - reports selection order of candidates
```

HDL generator:

```
----- FSM (XFC) to HDL converter - L(R)V - ver. O.y (99-08-..) -----

usage: xfc2hdl [<options>] XFC-file HDL-file [script_file]

  script_file == “dc_shell” script for Synopsys, default is “fsm_extr.script”

Options:
= design style:
  -vhd[l]       == generate VHDL description (use extension by default)
  -v            == generate Verilog description
  -merge        == merge FSM and data-path (separated by default)
  -no-reset     == do not generate reset sequence
  -async-reset  == generate asynchronously resettable registers
  -slice <nr>   == divide data-path into parallel slices
                  (<nr> states per slice, no slicing by default)
= suppress I/O buffering:
  -no-in    == do not generate input buffers
  -no-out   == do not generate output buffers
  -no-buff  == do not generate input/output buffers
= synthesis settings for ‘dc_shell’ script, defaults shown as “[...]”:
  -ungroup  == ungrouping all components (smaller and faster design
               but synthesis can be very time consuming)
  -flatten   == ungrouping even before FSM extraction (can be too expansive)
  -no-fsm   == no FSM extraction nor optimization
  -quit     == ‘quit’ command added to the end of the script
  -netlist <hdl>    == generate synthesized netlist also in VHDL or Verilog
                       (Synopsys data-base only by default)
  -clock <period>         == clock period                 [undefined]
  -area <gates>           == maximum total area           [undefined]
  -fsm_area <gates>       == maximum FSM area             [undefined]
  -power <value>          == maximum power consumption   [undefined]
  -low, -medium, -high    == mapping efforts              [medium]
  -auto, -binary, -gray, -one-hot    == FSM encoding type   [auto]
= additional options - HDL generating, debugging, etc.:
  -script-states == add state declarations also to the script
  -report-power  == add a command to report power consumption
  -blocknames    == dump XFC block names (as comments)
```

Appendix B. IRSYD - syntax in BNF

This document covers syntactical requirements of a system described in IRSYD and represented as a set of ASCII-files.

Keywords are as short as possible (IRSYD is not for a human being) and are represented using **bold** *font.*

Any construction can be separated by a white-space (space, tab, new-line, etc.), unless not defined otherwise. The grave accent (`) is the part of keywords, for instance.

1. System’s Structure

There are no reserved words. Built-in keywords, which can be mistaken for identifiers (names, types, etc.) because of their location, start with “‘”. In another places (e.g. structural elements), only predefined keywords can be used and there is no need to use special markings.

Any module, declaration, etc. is visible only in the module where it has been declared, and in the sub-modules and processes of that module.

Any file, containing description in IRSYD, can be viewed as a system (any subsystem is in principle a system).

The top level module corresponds to the whole system.

<file> ::= <system>

<system> ::= <module_body>

1.1. Modules

<module> ::= **mdl** <module_name> **{** <module_body> **}**

<module_body> ::= [<module_declarations> | <module> | <function> |
<process> | <import>]+

1.2. Functions

Function is a special case of module and therefore more or less the same syntax is used. It is described separately to underline the differences. There are two types of functions, which differ on closed-ness; i.e. can they access global variables. Access (read/write) is allowed by default. So called closed functions are used to model functions without side effects. These functions can access only data elements, which are described inside the function itself (parameters belong to the function). This means also that a pointer can be accessed but not data, pointed by that pointer, if the data exists outside the function. Hierarchical buildup of the internal structure (used by simu-

lator/synthesizer) must provide corresponding information. Functions are always called following the “call-by-value” model; i.e. parameters are copied and then passed to the function.

```
<function>     ::= mdl <function_name> <return_variable> [+]
                   { <function_header> <process> }
```

“+” defines closed function. Return variable is a variable defined in the process (<variable_list> of the <process>). Execution engine creates an internal (hidden) variable where the result is copied before deallocating variables. There is exactly one process in a function.

```
<function_header>
               ::= [ <function_declarations> | <function> ]+
```

1.3. Imports

For reusability purposes the mapping of the package’s name onto actual files (or libraries) is left for tools.

```
<import>        ::= imp { [<package_name>]* }

<package>       ::= [ <module_declarations> | <module> | <function> |
                      <process> | <import> ]*
```

Difference between <module_body> and <package> - <module_body> cannot be empty but <package> can (for debugging or what ever purposes).

2. Processes

```
<process>       ::= prc <process_name> { <process_body> }

<process_body>  ::= <process_declarations> <control_flow>
```

2.1. Control Flow

The order of flow nodes is irrelevant because of labels. Semantics requires that there must exist exactly one <entry_node>.

```
<control_flow>  ::= [ <flow_node> ]+

<flow_node>     ::= <entry_node> | <exit_node> | <condition_node> |
                    <operation_node> | <split_node> | <join_node> |
                    <activate_node>

<entry_node>    ::= entry <label> <next_node> [<attribute>]* ;

<exit_node>     ::= exit <label> [<attribute>]* ;

<condition_node>
                ::= cnd <label> <variable_name>
                    [<next_node> <case_value>]+ [<attribute>]* ;
```

Variable (<variable_name>) must be of type boolean or set. Number of transitions (<next_node>) depends on the number of possible values of the variable.

```
<case_value>    ::= <boolean_value> | <set_element> | *
    "*" is equal to default (or others, etc.)

<operation_node>
                ::= op <label> [<statement>]* <next_node>
                    [<attribute>]* ;

<split_node>    ::= split <label> [<next_node>]+ [<attribute>]* ;

<join_node>     ::= join <label> <next_node> [<attribute>]* ;

<activate_node> ::= act <label> <module_name>
                    ( [<port_reference> [+]]* )
                    <next_node> [<attribute>]* ;
    "+" denotes call-by-value, i.e. caller makes copy. Default is
    call-by-reference. Original and copy are never synchronized -
    call-by-reference must be used to pass values back.

<next_node>     ::= <next_label> [<event>]

<port_reference>
                ::= <port_name> | <variable_name> |
                    <shared_variable_name> | <constant_name>
    Both type of variables (declared in <module_declarations> and
    <process_declarations>) are allowed. Potential conflicts when
    using simple variables and constants can be solved combining
    port access modes.
```

2.2. Statements

```
<statement>     ::= [ <result> <expression> [<attribute>]* ]

<result>        ::= <port_name> | <variable_name> |
                    <shared_variable_name> | `nil
    "`nil" as a result means that the result is ignored - <array_write>,
    <pointer_write>, <send_operation>, <receive_operation>.
```

2.3. Events

```
<event>         ::= `( <expression> [<attribute>] )
    An expression with boolean return type is required.
```

3. Declarations

```
<module_declarations>
                ::= [ <type_list> | <port_list> | <constant_list> |
                      <shared_variable_list> | <attribute> ]*

<function_declarations>
                ::= [ <type_list> | <port_list> | <constant_list> |
                      <attribute> ]*
    All port modes are allowed and function is called making copies
    of all parameters, i.e. function call (in <expression>) is
    equivalent to call-by-value for all parameters.
```

```
<process_declarations>
                ::= [ <type_list> | <constant_list> |
                      <variable_list> | <attribute> ]*

<attribute>     ::= @ <attribute_name> [ ( [<attribute_value>]* ) ]
    In ERE  “@ *[a-zA-Z_][0-9a-zA-Z_]*”
```

3.1. Types

```
<type_list>     ::= type { [ <type_declaration> | <attribute> ]* }

<type_declaration>
                ::= <type_name> : [<array_declaration>]*
                    <data_type> [<attribute>]* ;

<data_type>     ::= <type_name> | <set_type> | <record_type> |
                    <builtin_type>

<set_type>      ::= `set { [<set_element>]+ }

<record_type>   ::= `rec { [<record_fields>]+ }

<record_fields> ::= [<field_name>]+ : [<array_declaration>]*
                    <data_type> [<attribute>]* ;

<builtin_type>  ::= <boolean_type> | <integer_type> | <string_type> |
                    <signed_type> | <unsigned_type> | <pointer_type> |
                    <module_type>

<boolean_type>  ::= `bool

<integer_type>  ::= `int [<integer_range>]
    Default is ‘-- ++’, i.e. full range.

<string_type>   ::= `str [<vector_length>]
    Default is ‘0 ++’ (or the actual length).
    The same applies also for <signed_type> and <unsigned_type>.

<signed_type>   ::= `sgn [<vector_length>]
    The most-significant-bit is the left-most bit.
    Two’s complementary logic is used.

<unsigned_type> ::= `usgn [<vector_length>]
    The most-significant-bit is the left-most bit.

<vector_length> ::= <integer_range> | `nil
    “`nil” is used for (initially) empty strings.

<pointer_type>  ::= `pnt <data_type>

<module_type>   ::= `mdl

<array_declaration>
                ::= `arr <array_range>

<array_range>   ::= <integer_range> | <data_type>
    <data_type> of <integer_type> is equivalent to <integer_range> and
    allows both bounded & unbounded arrays;  <set_type> (and
    <boolean_type>) gives bounded arrays; and all other types represent
    unbounded associative arrays in practice.
```

3.2. Constants

```
<constant_list> ::= const { [ <constant_declaration> | <attribute> ]* }

<constant_declaration>
                ::= [<constant_name>]+ : [<array_declaration>]*
                    <data_type> = <constant_values> [<attribute>]* ;

<constant_values>
                ::= <constant_array_values> | <constant_value>

<constant_array_values>
                ::= { [<constant_values> [<repeat>]]+ [<attribute>]* }

<repeat>        ::= ` [<positive_number> | *]
    In ERE  “` *[0-9]+” or “` *\*”

<constant_value>
               ::= <set_element> | <record_value> | <boolean_value> |
                   <decimal_number> | <string> | <bit_vector> |
                   <identifier_name> | <module_name> |
                   <function_name> | <expression>
    Legal <constant_value> depends on the type of the constant
    <set_type>     -> <set_element>
    <record_type>  -> <record_value>
    <boolean_type> -> <boolean_value>
    <integer_type> -> <decimal_number>
    <string_type>  -> <string>
    <signed_type>, <unsigned_type> -> <bit_vector>
    <pointer_type> -> <identifier_name>
    <module_type>  -> <module_name>, <function_name>
    Calculatable value (<expression>) is calculated when the module is
    activated. Usage of non-existing values results with ‘`nil’, i.e.
    undefined constant value (an erroneous situation)

<record_value>  ::= { [<constant_values>]+ [<attribute>]* }

<identifier_name>
                ::= <constant_name> | <port_name> |
                    <shared_variable_name> | <variable_name>
```

3.3. Ports, Variables

```
<port_list>     ::= port { [ <port_declaration> | <attribute> ]* }

<port_declaration>
                ::= [<port_name>]+ : <port_mode>
                    [<array_declaration>]* <data_type>
                    [= <variable_values>] [<attribute>]* ;

<port_mode>     ::= in | out | io
```

```
<shared_variable_list>
                ::= shvar { [ <shared_variable_declaration> |
                              <attribute> ]* }

<shared_variable_declaration>
                ::= [<shared_variable_name>]+ : [<array_declaration>]*
                    <data_type> [= <variable_values>] [<attribute>]* ;

<variable_list> ::= var { [ <variable_declaration> | <attribute> ]* }

<variable_declaration>
                ::= [<variable_name>]+ : [<array_declaration>]*
                    <data_type> [= <variable_values>] [<attribute>]* ;

<variable_values>
                ::= <constant_values> | `nil
```

*“`**nil**” marks undefined value as is also the default value.*

4. Expressions

All expressions are represented in prefix format to avoid usage of parentheses. Execution engine allocates and deallocates all needed temporary variables (depending on the type). In explanations the traditional algebraic notation is used because of the readability (keywords/operations without “`”).

Temporary results have always large enough range - adjustments not needed. No exceptions here (i.e. over-/under-flow). Simulators/synthesizers must detect infinite (or unreasonably(sic!) large) values and react to avoid system crashes.

Before assigning values to the target the range of the result is adjusted depending on the target’s type:
I) <integer_type> based types (also array elements with corresponding indexing):
1) No adjustment when result is in the target’s range;
*2) Actual result is larger than target (i.e. out of range) - result is “`**nil**” (impossible to adjust).*
II) <string_type>, <signed_type> and <unsigned_type> based types, and array slices (of arrays with <integer_type> indexing):
*Reference point is always “**lo()**” (lowest) and elements are assigned towards the “**hi()**”. The result is adjusted in the “**hi()**” end when needed:*
1) Target has wider range than result (hi(target)-lo(target) > hi(result)-lo(result)) - expanding result. Values of elements with indexes “hi(result)+1” to “hi(target)” depends on the type:
a) <string_type> - “\000”;
b) <unsigned_type> - “0”;
c) <signed_type> - value of hi(result), i.e. sign-extension (applies actually also for the <unsigned_type>); and
*d) array elements - “`**nil**”, i.e. undefined value.*
2) Ranges are equal (hi(target)-lo(target) = hi(result)-lo(result)) - no adjustment needed.
3) Target has narrower range than result (hi(target)-lo(target) < hi(result)-lo(result)) - truncating result. Only elements with indexes “lo(target)” to “hi(target)” are used.

*Built-in type conversion is supported only between <integer_type>, <string_type>, <signed_type> and <unsigned_type>. The first ("fr()") element of a string is used, full values for other types. All cross-conversions are based on the integer interpretation of a value - 0 .. 255 for an element of a string, 0 .. 2**(hi()-lo()+1)-1 for <unsigned_type>, and -2**(hi()-lo()) .. 2**(hi()-lo())-1 for <signed_type>.*

All other conversions depend on the interpretation of types and are not defined here. Corresponding functions must be provided by the front-end compilers, which have interpretation for the rest of the data types.

```
<expression>    ::= <operand> | <operation>

<operation>     ::= <arithmetic_operation> | <logic_operation> |
                    <equality_operation> | <relational_operation> |
                    <string_operation> | <shift_operation> |
                    <access_operation> | <special_operation> |
                    <function_call>

<operand>       ::= `nil | <expression> | <constant_name> |
                    <port_name> | <shared_variable_name> | <variable_name> |
                    <boolean_value>
```

*"`**nil**" marks undefined value (of any type).*
Only identifier names are allowed to avoid type conflicts when using constant values. Type of operands must match with required types. Some operations allow to use constants directly - when only certain type is allowed. No type checking at run time (only compilation). Different subtypes are adjusted depending on the operation types.
<boolean_value> is allowed for operands of <boolean_type>.

4.1. Arithmetic Operations

Defined for <integer_type>, <signed_type> and <unsigned_type>. Both operands and result must be of the same type.
Pointer arithmetic is memory architecture depending and therefore arithmetic operations are not defined for <pointer_type>. The same applies for comparisons (see 4.3. Relational Operations)

```
<arithmetic_operation>
                ::= <b-arithmetic_operation> | <u-arithmetic_operation>

<b-arithmetic_operation>
                ::= <b-arithmetic_operator> <operand> <operand>

<arithmetic_operator>
                ::= + | - | * | / | `md | `dvn | `mdn
```

Because division of integers is not exactly defined for some languages, C and C++ for instance, two different division operations are defined, together with corresponding modulus operations:

1. *Definition of '/' is based on assumption that |A/B|=|A|/|B| - used for real numbers - absolute value of the quotient is truncated and the correct sign is assigned to the result.*
   An example - -23 / 4 = -5 ;
2. *"`**dvn**" is defined as "the largest integer equal to or less than A/B (dividing real numbers)", i.e. _|A/B|_ .*
   An example - -23 dvn 4 = -6 ;
3. *Modulus is defined for both divisions by "A = (A/B)*B + A mod B"*

*where “***`md***” uses “***/***” and “***`mdn***” uses “***`dvn***” operations.*
Examples - -23 md 4 = -3 and -23 mdn 4 = 1 .
*“***`md***” corresponds to “rem” and “***`mdn***” to “mod” in VHDL and ADA.*

More examples:

```
  7 /  5 =  1           7 md  5 =  2
 -7 /  5 = -1          -7 md  5 = -2
  7 / -5 = -1           7 md -5 =  2
 -7 / -5 =  1          -7 md -5 = -2

  7 dvn  5 =  1         7 mdn  5 =  2
 -7 dvn  5 = -2        -7 mdn  5 =  3
  7 dvn -5 = -2         7 mdn -5 = -3
 -7 dvn -5 =  1        -7 mdn -5 = -2
```

<u-arithmetic_operation>
::= <u-arithmetic_operator> <operand>

<u-arithmetic_operator>
::= **`pl** | **`ng**

4.2. Logic Operations

Defined for <boolean_type>, <signed_type> and <unsigned_type>. Both operands and result must be of the same type.

<logic_operation>
::= <b-logic_operation> | <u-logic_operation>

<b-logic_operation>
::= <b-logic_operator> <operand> <operand>

<b-logic_operator>
::= **&** | **|** | **!&** | **!|** | **`x**

<u-logic_operation>
::= <u-logic_operator> <operand>

<u-logic_operator>
::= **!**

4.3. Equality Operations

Defined for all types, including user defined.
Both operands must be of the same type. Result is <boolean_type>.

<equality_operation>
::= <equality_operator> <operand> <operand>

<equality_operator>
::= **=** | **/=**

4.4. Relational Operations

Defined for <integer_type>, <signed_type> and <unsigned_type>.

Both operands must be of the same type. Result is <boolean_type>.

```
<relational_operation>
                ::= <relational_operator> <operand> <operand>

<relational_operator>
                ::= < | > | <= | >=
```

4.5. String Operations

Defined for <string_type>, <signed_type> and <unsigned_type>.

```
<string_operation>
                ::= <concatenate_operation> | <slice_operation>

<concatenate_operation>
                ::= `conc <operand> <operand>
```

Both operands and result must be of the same type.

```
<slice_operation>
                ::= `slc <first_element> <slice_length> <source_vector>
```

*<first_element> and <slice_length> define which part of the <source_vector> results. Direction is from always from left to right (NB! positive <slice_length>). Only elements from the <source_vector> can be used and therefore the result’s length can be equal to <vector_length>, to(<source_vector>) - <first_element> + 1 , or “`**nil**” in the extreme case.*

```
<first_element> ::= <decimal_number>

<slice_length>  ::= <positive_number>
```

4.6. Shift Operations

Defined for <signed_type> and <unsigned_type>.
Shift operations are not defined for <integer_type> because the behavior depends on the representation (representation is not defined).

```
<shift_operation>
                ::= <logic_shift_left> | <logic_shift_right>
                    <arithmetic_shift_left> | <arithmetic_shift_right>

<logic_shift_left>
                ::= `lsl <source_vector> <shift_count>

<logic_shift_right>
                ::= `lsr <source_vector> <shift_count>

<arithmetic_shift_left>
                ::= `asl <source_vector> <shift_count>

<arithmetic_shift_right>
                ::= `asr <source_vector> <shift_count>

<shift_count>   ::= <operand> | <decimal_number>
```

The <operand> (result of <expression>) must be of <integer_type>.

```
    Negative result applies shift in another direction.

<source_vector> ::= <operand>
```

4.7. Access Operations

Defined for arrays and <pointer_type>. <array_slice_read> and <array_slice_write> are defined for arrays with integer address range only. The first operand of write operations is the data to be stored. Assigning “`nil” value to an element of an associative array removes that element from the array.

```
<access_operation>
               ::= <array_read> | <array_write> | <array_slice_read> |
                   <array_slice_write> | <pointer_read> | <pointer_write>

<array_read>   ::= `rd <array_name> <index_count> [<index>]*
    Return type depends on the number of used indexes (dimensions) -
    “`rd <array_name>” is equivalent to “<array_name>”.

<array_write>  ::= `wr <operand> <array_name> <index_count> [<index>]*
    Type of <operand> depends on the number of used indexes.
    Returns stored value and return type is the type of <operand>.

<array_slice_read>
               ::= `rds <array_name> <first_address> <last_address>
    Return type is “`arr <first_address> <last_address>”

<array_slice_write>
               ::= `wrs <operand> <array_name> <first_address>
                   <last_address>
    Returns stored value and return type is the type of <operand>.
    <array_name> must be of type “`arr <integer_range>” with a range
    equal or larger than defined by <first_address> and <last_address>.
    <operand> must be of type “`arr <integer_range>” with a range
    equal to defined by <first_address> and <last_address>.

<array_name>   ::= <operand>
    Must be defined as an array (of any type).

<index>        ::= <identifier_name>

<index_count>  ::= <positive_number>
    Needed because of the prefix format - the number of operands must be
    known (at compile time?).

<first_address>, <last_address>
               ::= <identifier_name> | <decimal_number>

<pointer_read> ::= `rd <pointer_name>
    Access to type “`pnt <data_type>” returns type “<data_type>”.

<pointer_write> ::= `wr <operand> <pointer_name>
    Access to type “`pnt <data_type>” requires “<data_type>”.
    Returns stored value and return type is the type of <operand>.
```

```
<pointer_name>  ::= <operand>
        Must be defined as a pointer (of any type).
```

4.8. Special Operations

The special operations simplify handling of complex data types. In most of the languages, special built-in functions are used for that purpose, but in IRSYD, they are classified as operations because of their syntax (prefix notation).

```
<special_operation>
                ::= <boundary_operation> | <new_operation> |
                    <delete_operation> | <idle_operation> |
                    <send_operation> | <receive_operation> |
                    <buffer_size> | <index_list>

<boundary_operation>
                ::= <boundary_operator> <boundary_operand>
    Defined for <integer_type>, <signed_type>, <unsigned_type> and
    arrays with <integer_range>. Returns <integer_type>.

<boundary_operator>
                ::= `lo | `hi | `fr | `to
    “Low”, “high”, “form” and “to” boundaries of types’ range.
    Defined for <integer_type>, <signed_type>, <unsigned_type> and
    for arrays of <integer_range>.
    lo() = fr() and hi() = to()  when  fr() <= to() ,  and
    hi() = fr() and lo() = to()  when  fr() >= to() ,  where
    fr() = <from_value> and to() = <to_value>.

<boundary_operand>
                ::= <identifier_name> | <type_name>

<new_operation> ::= `new <type_name>
    Type <type_name>. Allocates structure needed to represent
    corresponding data type. Allocation of variables of activated
    modules should use the same operation (to avoid conflicts
    of structures). The same should apply for deallocation.

<delete_operation>
                ::= `del <operand>
    Deallocating structure allocated by <new_operation>. <operand>
    must be of a pointer type.

<idle_operation>
                ::= `idle
    Type “`bool”. Returns value “`t” when all active threads of the
    system have been paused, “`f” otherwise.

<send_operation>
              ::= `snd <message> <channel_id> <message_index> <replace_flag>
    Sends <message> into <channel>. Replaces message <message_index>
    (<replace_flag> is “`t”) or inserts before it (<replace_flag> is
    “`f”). Return type is “`bool”. Returns value “`t” on success,
    “`f” otherwise. Fails when <message_index> < 0, or on attempt to
    replace a non-existing message, or to create a gap between messages.
```

```
<receive_operation>
               ::= `rcv <message> <channel_id> <message_index> <remove_flag>
    Reads a message from <channel> from location <message_index>.
    Removes when <remove_flag> is “`t” (non-depending on the type
    correctness). Stores received message into <message>. Return type
    is “`bool”. Returns value “`t” on success, “`f” otherwise. Fails when
    <message_index> < 0, or on attempt to remove a non-existing message,
    or to receive a message of wrong type.

<message>       ::= <operand>
    Any type is possible. Even different messages of different type can
    be sent to the same channel.

<buffer_size>   ::= `bfsz <channel_id>
    Returns current buffer size of channel <channel_id>, or -1 when
    channel has not been created.

<channel_id>    ::= <operand> | <decimal_number>
    Must be of <integer_type>.

<message_index> ::= <operand> | <positive_number>
    Must be of <integer_type> and have a positive value.

<replace_flag>, <remove_flag>
                ::= <operand>
    Must be of <boolean_type>.

<index_list>    ::= `lst <array_name>
    Defined for associative arrays. Returns list of indexes with type
    integer range array of addressing type.
```

4.9. Function Calls

User defined functions are defined as special modules (see 1.2. Functions). Return type is defined in the declaration.

```
<function_call> ::= `act <function_name> <operand_count> [<operand>]*
    <operand_count> should be known beforehand. This does not contradict
    with other requirements for IRSYD - no operator/function overloading.
    Function calls with variable number of operands should be organized
    in different manner (stacks, i.e. arrays).

<operand_count> ::= <positive_number>
```

5. Atomic Objects

```
<constant_name>, <port_name>, <shared_variable_name>, <variable_name>
                ::= <name> | <structured_name>

<structured_name>
                ::= <name> [ ’ <field_name> ]+

<module_name>, <attribute_name>, <set_element>, <field_name>,
<function_name>, <label>, <next_label>, <package_name>, <process_name>,
<return_variable>, <type_name>
               ::= <name>
```

```
<attribute_value>
                ::= <string>

<integer_range> ::= <from_value> <to_value>

<from_value>, <to_value>
                ::= -- | ++ | <decimal_number> | <constant_name>
    “--” and “++” denote negative and positive infinity correspondingly

<name>          ::= [a-zA-Z_][a-zA-Z0-9_]*

<boolean_value> ::= `t | `f

<decimal_number>
                ::= [-+] <positive_number>

<positive_number>
                ::= [0-9]+

<string>        ::= ”[<character>]*”

<character>     ::= ‘ ’ | ! | \” | # | ... | [ | \\ | ] | ... |
                    ~ | <byte_code>
    Visible ASCII codes only (from 32 to 126).
    Extended character sets (e.g. ISO 8859-1) are not supported to
    avoid confusions between different sets.

<byte_code>     ::= \000 | ... | \377
    All 8-bit codes from 0 to 255.

<bit_vector>    ::= ”[<bit_value>]+”

<bit_value>     ::= 0 | 1 | -
    IEEE-1164 etc. should use ‘set’ types.
```

6. Comments

Comments in IRSYD are totally ignored. Comments from input languages should be transferred into attributes if they are viewed as part of the documentation of a system.

```
<comment>       ::= // <character>[*]
    In ERE  “//.*$” , i.e. anything from ‘//’ to the end of the line.
```

Appendix C. IRSYD - C++ class structure

Notation

```
<object> ::= <declaration>      - <object> is defined as <declaration>

{ <element> ... }               - structure

<element_name> : <type>

&& <object>                     - hard reference (pointer) onto <object>,
                                  solved during building

& <object>                      - soft reference onto <object> - consists of
                                  name and direct reference, solved after
                                  building (linking phase)

<type> [n]                      - unordered list of <type> (‘n’ - exactly n,
                                  ‘*’ any number, ‘+’ at least one element)

<type> [[n]]                    - ordered list of <type> (‘n’ - exactly n,
                                  ‘*’ any number, ‘+’ at least one element)

<type> @[n]                     - unordered attributed list of <type>
                                  (‘n’ - exactly n, ‘*’ any number,
                                   ‘+’ at least one element)

<type> @[[n]]                   - ordered attributed list of <type>
                                  (‘n’ - exactly n, ‘*’ any number,
                                   ‘+’ at least one element)
```

Ordinary lists are derived from <object>,
attributed lists from <attributed_object>.

```
<object> +                      - derived from <object>

( <choice_1>, ... )             - choice

@ <comment> @
```

0. Basic Declarations

```
boolean ::=             ( False, True )

string ::=              ( 0, ..., 255 ) [*]     @ implementation depending! @

import_flag ::=         ( False, string )        @ reference onto package @
```

```
{ & <object> } ::= {
  name :              string                 @ solved during compilation @
  pointer :            && <object>            @ solved during linking @
}

object ::= {                                  @ abstract @
  name :               string
  parent :             && object
  type :               ( Attribute, Module, Function, ClosedFunction,
                         Process, Successor, Event, Statement, EntryNode,
                        ExitNode, ConditionNode, OperationNode, SplitNode,
                         JoinNode, ActivateNode, PortReference, Type,
                         Value, AtomicValue, Port, Constant,
                         SharedVariable, Variable, ConstantParameter,
                         VariableParameter, Expression, FunctionCall,
                         UnorderedList, OrderedList,
                         UnorderedAttributedList, OrderedAttributedList )
}
```

{Unordered|Ordered}[Attributed]List - templates of list are also derived.

```
attribute ::= object + {
  values :             string [*]
  imported :           import_flag
}

attributed_object ::= object + {              @ abstract @
  attributes :         attribute [*]
}
```

1. System’s Structure

```
module ::= attributed_object + {
  imports :            string [[*]]        @ list of imported packages @
  types :              type @[*]
  ports :              port @[[*]]
  constants :          constant @[*]
  shared_variables :   shared_variable @[*]
  modules :            module [*]
  functions :          function [*]
  processes :          process [*]
  imported :           import_flag
}

function ::= attributed_object + {
  types :              type @[*]
  ports :              port @[[*]]
  constants :          constant @[*]
  functions :          function [*]
  function_body :      process
  return_variable :    & variable         @ a variable in function_body @
  imported :           import_flag
  closed_function :    boolean
}
```

2. Processes

```
process ::= attributed_object + {
  types :               type @[*]
  constants :           constant @[*]
  variables :           variable @[*]
  control_flow :        flow_node [*]
  imported :            import_flag
}

successor ::= object + {
  next_node :           & flow_node
  case_value :          base_value
  event :               event
}

event ::= attributed_object + {
  expression :          ( False, base_expression )
}

statement ::= attributed_object + {
  target_variable :     & data_element
  expression :          base_expression
}

flow_node ::= attributed_object + {             @ abstract @
  successors :          successor [n]           @ ‘n’ defined by type @
  predecessors :        { & flow_node } [*]
}

entry_node ::= flow_node + {}                   @ n == 1 @

exit_node ::= flow_node + {}                    @ n == 0 @

condition_node ::= flow_node + {                @ n >= 2 @
  condition :           & data_element
}

operation_node ::= flow_node + {                @ n == 1 @
  statements :          statement [*]
}

split_node ::= flow_node + {}                   @ n >= 2 @

join_node ::= flow_node + {}                    @ n == 1 @

activate_node ::= flow_node + {                 @ n == 1 @
  module :              & module
  ports :               port_reference [[*]]
}

port_reference ::= object + {
  port :                & data_element
  copy :                boolean
}
```

3. Declarations

```
type ::= attributed_object + {
  sub_types :           { & type } [[*]]        @ composed types @
  dimension :           ( False, & type )       @ array addressing @
  range :               base_value [[*]]      @ set-elements / boundaries @
  builtin_type :        ( User, Record, Set, Boolean, Integer, String,
                          Signed, Unsigned, Pointer, Module, Array )
  imported :            import_flag
}
```

sub_types == [[1]] - User, Array; [[+]] - Record; empty - others.
dimension == False - scalar (builtin_type not Array).
range - list for set types, 2 values otherwise (left & right).

```
base_value ::= object + {                       @ abstract @
  type :                & type
}

atomic_value ::= base_value + {
  value :               ( False, string )       @ False == `nil @
}
```

‘type’ -- non-composite type

```
value ::= base_value + {
  value :               base_value [[*]]
}

data_element ::= attributed_object + {          @ abstract @
  initial_value :       base_value
  value :               base_value
}
```

‘type’ can be skipped because value has it. Kept for safety.

```
port ::= data_element + {
  mode :                ( In, Out, Io )
  imported :            import_flag
}

constant ::= data_element + {                   @ initial value == value @
  imported :            import_flag
}

shared_variable ::= data_element + {
  imported :            import_flag
}

variable ::= data_element + {}
```

4. Expressions

```
base_expression ::= object + {}                 @ abstract @

constant_parameter ::= base_expression + {
  value :               base_value
}
```

```
variable_parameter ::= base_expression + {
  parameter :           & data_element
}

expression ::= base_expression + {
  operator :            (@ arithmetic @
                          Add, Subtract, Multiply, Divide, Modulus,
                          Divide2, Modulus2, Plus, Minus,
                         @ logic @
                          And, Or, Nand, Nor, Xor, Not,
                         @ equality @
                          Equal, NotEqual,
                         @ relational @
                          Less, Greater, NotGreater, NotLess,
                         @ string @
                          Concatenate, Slice,
                         @ shift @
                          LogicLeft, LogicRight, ArithmLeft, ArithmRight,
                         @ access @
                          ArrayRead, ArrayWrite, ArraySliceRead,
                          ArraySliceWrite, PointerRead, PointerWrite
                         @ functions @
                       LowFn, HighFn, FromFn, ToFn, NewFn, DeleteFn, IdleFn,
                          Send, Receive, BufferSize, IndexList )
  parameters :          base_expression [[*]]
}

function_call ::= base_expression + {
  function :           & function
  parameters :         base_expression [[*]]
}
```

Number of operands is the size of parameters list.

5. Derivation Hierarchy

```
  { class }     - hidden class (abstract, etc.)

  [ class ]     - visible class (including virtual)

  < template >  - template (for list)


{ object }
  +---> [ attribute ]
  +---> < unordered_list >
  +---> < ordered_list >
  +---> { attributed_object }
  |       +---> < unordered_attributed_list >
  |       +---> < ordered_attributed_list >
  |       +---> [ module ]
  |       +---> [ function ]
  |       +---> [ process ]
  |       +---> { flow_node }
  |       |       +---> [ entry_node ]
  |       |       +---> [ exit_node ]
  |       |       +---> [ condition_node ]
  |       |       +---> [ operation_node ]
  |       |       +---> [ split_node ]
  |       |       +---> [ join_node ]
  |       |       +---> [ activate_node ]
  |       +---> [ event ]
  |       +---> [ statement ]
  |       +---> [ type ]
  |       +---> { data_element }
  |               +---> [ port ]
  |               +---> [ constant ]
  |               +---> [ shared_variable ]
  |               +---> [ variable ]
  +---> { base_expression }
  |       +---> [ constant_parameter ]
  |       +---> [ variable_parameter ]
  |       +---> [ expression ]
  |       +---> [ function_call ]
  +---> { base_value }
  |       +---> [ atomic_value ]
  |       +---> [ value ]
  +---> [ successor ]
  +---> [ port_reference ]
```