HARDWARE MAPPING AND MODULE SELECTION IN THE HYPER SYNTHESIS SYSTEM

by

Chi-Min Chu

Memorandum No. UCB/ERL M92/46

8 May 1992

HARDWARE MAPPING AND MODULE SELECTION IN THE HYPER SYNTHESIS SYSTEM

by

Chi-Min Chu

Memorandum No. UCB/ERL M92/46

8 May 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

HARDWARE MAPPING AND MODULE SELECTION IN THE HYPER SYNTHESIS SYSTEM

by

Chi-Min Chu

Memorandum No. UCB/ERL M92/46

8 May 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Hardware Mapping and Module Selection in the HYPER Synthesis System

by

Chi-Min Chu

Abstract

The computationally intensive parts of high-performance real-time systems, such as video, image or speech recognition, are usually implemented on clusters of heavily pipelined data paths, controlled by a relatively simple finite state machine. Synthesizing such architectures is a tough task for a human designer and adequate CAD tools are therefore a must. The HYPER synthesis system has been developed to address this class of architectures.

The focus of this dissertation is on the interface between HYPER and the hardware. This platform includes hardware selection, hardware mapping, and hardware database. The task of the hardware selection is to select a set of hardware modules which minimize the implementation cost of an algorithm, given the timing and throughput constraints. At the same time, simple operators are clustered into large combinatorial blocks to reduce the register count and to increase the throughput. The proposed approach is organized as a search employing a relaxed scheduling for cost estimation and uses a simple, yet accurate timing analysis to verify timing constraints.

Hardware mapping is to translate a flow graph with scheduling and allocation data into an actual hardware structure, consisting of data paths, a central controller, and interface logic. The major tasks of the hardware mapping involve data path optimizations, such as data path partitioning, multiplexer reduction and register file merging, and control path optimizations. The goal is to improve the area utilization under the throughput constraints.

Both the hardware selection and the hardware mapping processes require the information about the available cell library. A rule-based library database with many useful access routines was therefore developed to provide the information. Currently, the HYPER hardware database implements the Lager cell library, which includes three types of modules: data-path modules, array modules, and standard-cell modules.

Many real-time applications have been synthesized using HYPER, including a

Viterbi Processor, a 7th order IIR filter and a CORDIC processor. Simulations have been performed to verify the correctness of these designs and layouts have been produced to demonstrate the quality of the transformations in the hardware mapper. Area-delay trade-offs of different implementations have also been made to study various design requirements. Based on the study and critique of the final layouts of many designs, the hardware platform has been gradually refined.

Jan Rabaey
Committee Chairman

Contents

List of Figures

List of Tables

Acknowledgements

I would like to acknowledge the following people and organizations for their help to the successful completion of this project.

Professor Jan Rabaey, my research advisor, deserved many credits of this project. He not only first defined the HYPER framework, but also worked with me throughout this project. His leadership of the HYPER Group is superb. His guidance, support and inspiration is invaluable and it has been a great pleasure to work with him.

The other members of the HYPER Group contribute many useful suggestions to this project. Miodrag Potkonjak worked on the HYPER scheduling, allocation, assignment, and transformations. He was an algorithm expert and often provided me with many interesting ideas. Phu Hoang has been working on the Silage to flow graph compiler and the flow graph to C transformer. He also helped in defining many ASCII flow graph formats. His useful comments and fresh perspectives about the project have been of great help. Markus Thaler, one early member of HYPER, defined Turtle, the first language adopted by HYPER, and suggested several interesting ideas. David Schultz performed an extensive Thor simulation for the IIR example to verify the correctness of HYPER.

Many colleagues of mine also provided their personal help: Mani Srivastava has been of great help for sharing his expertise in Lager and many other areas. Sam Sheng provided many assistance in making the Thor simulation work. Ken Rimey and Ed Wang helped in solving many problems in running Lisp.

I would like to thank Professor Newton and Professor Stone for being the second reader and the third reader of this dissertation.

This project is supported by DARPA under various contract numbers. Harris Semiconductor Inc., the industrial supporter of HYPER, provides part of the HYPER funding and gives us many useful feedbacks in improving the system.

Special thanks to my colleagues Bernard Shung, Alex Lee, Dev Chen, and my wife, Jeanne Tsai, for their cheerful supports.

Chapter 1

Introduction

1.1 High Level Synthesis and Existing Systems

Given a specification of the system behavior and a set of constraints, the task of high level synthesis is to find a structure that implements the required behavior while satisfying the constraints. The constraints may include timing, area, or power requirements of the final design. The *behavior* of a system is defined as the way the system interacts with its environment. The *structure* refers to the set of interconnected components that make up the system. Good overviews of high level synthesis and the state of art techniques can be found in [65] and [8].

The major tasks involved in high level synthesis are input compilation, algorithmic and architectural simulations, hardware module selection, resource estimation, hardware allocation, assignment, scheduling, transformation and hardware mapping. Each task can be further divided into subtasks. Many of the tasks are NP-hard problems [1]. Furthermore, they are interrelated and dependent on one another. Due to the complexity of the high level synthesis tasks, many high level synthesis systems have been developed to help human designers to find an efficient solution. Even with the aid of these synthesis systems, the synthesis task still requires many design iterations and interactions between the human designers and the synthesis environment. However, the design time is greatly reduced.

Most of the high level synthesis systems aim for a particular application and a

[1] NP is the set of all problems which can be solved by nondeterministic algorithms in polynomial time. NP-complete problems are a subset of the NP problems which cannot be solved by an deterministic algorithm in polynomial time[71].

target architecture. For example, IMEC's Cathedral II system [26] aims for a subset of digital signal processing (DSP) algorithms to be architecturally realized by a set of concurrent dedicated bit-parallel processors on a single chip. The System Architect's Workbench [22] [9] from Carnegie Mellon University supports three different synthesis paths: a general synthesis path, a pipelined-instruction-set-specific synthesis path, and a microprocessor specific synthesis path. There are also many other systems such as the ADAM system [37] from University of Southern California and the HAL system [49] from Carleton University. Most of these systems study some specific synthesis problems such as scheduling and estimation. A survey of the important current high level synthesis systems is presented in [74], which lists many of the major contributions and approaches of those systems.

Similar to most of the high level synthesis systems, The **HYPER** (which stands for high performance) [19] [29] [50] [59] synthesis system was developed to address the architectures of a particular type of applications – the computationally intensive parts of high-performance real-time systems, such as video, image or speech recognition systems. These systems are usually implemented on clusters of heavily pipelined data paths, controlled by a relatively simple finite-state machine. The amount of resource sharing is limited.

The most important feature of HYPER, which distinguishes it from other synthesis systems, is a single global optimization strategy. This optimization strategy drives not only the resource assignment and scheduling process but also the preceding optimizing transformations. In all those phases of the synthesis process, HYPER attempts to optimize a so called *resource utilization table*, which maps the utilization of the different resources (such as execution units, memory, interconnect and I/O bandwidth) onto the allotted time period.

1.2 Interface Between HYPER and Hardware

The focus of this dissertation is on the interface between HYPER and hardware. This hardware platform includes three modules: hardware selection, hardware mapping and hardware database. The system overview of HYPER is shown in Figure 1.1, which illustrates the major components of HYPER and how the platform is organized and interacts with the other modules of HYPER.

Hardware selection is the first step in the HYPER synthesis process. It links every operator in the flow graph to a hardware library element. At the same time, groups

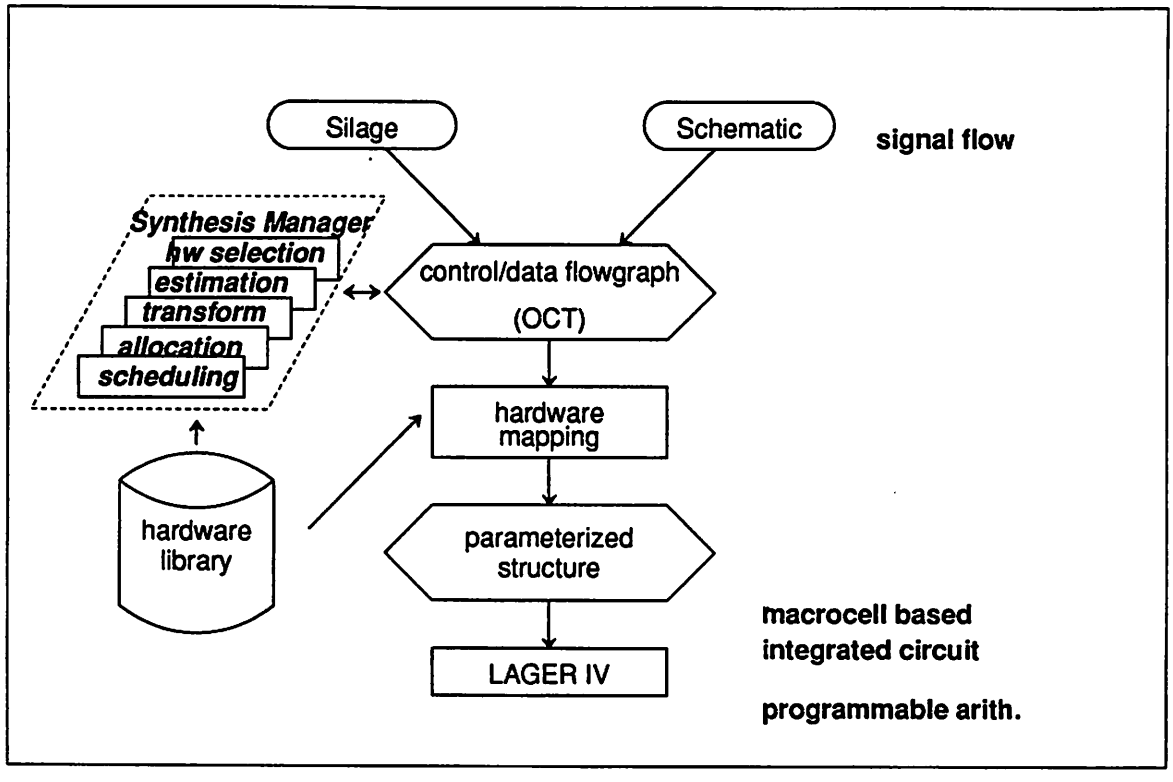

Figure 1.1: HYPER Overview

of operations are clustered into "composite" hardware nodes (which contain no registers), based on a careful timing analysis. From this point on, the synthesis tools can use the abstract *control step* or *clock cycle* concept, instead of having to operate on absolute time.

The result of the synthesis operations, such as the hardware assignment and scheduling information, is back-annotated onto the flow graph database. Once the synthesis process is completed, the hardware mapper translates the flow graph into a hardware structure containing data paths, a central control, and glue logic. Silicon can then be generated from the hardware structure using Lager IV [20] or other silicon compilation systems.

Both the hardware selection and the hardware mapping processes require an accurate knowledge of the available cell library in terms of functionality, speed, area, and black box view. This is provided by a rule-based library database. This library currently has more than thirty blocks which provide all the basic functions supported by HYPER. A database editor which helps designers to input, retrieve, edit or delete a cell and/or its attributes has also been developed. This editor allows the designers to easily introduce special purpose blocks.

The motivation of this project can be summarized as follows:

- *Compare tradeoffs between various hardware implementations.* Both the hardware selection and the hardware mapping processes can produce several designs for the same specification depending on the given constraints. This facilitates the designers in exploring the design space and comparing the tradeoffs among area, speed, power etc.

- *Verify synthesis results through real hardware.* Through the hardware platform, the performance of the synthesis processes, including scheduling, allocation, assignment and transformation, can be accurately quantified by mapping to real hardware.

- *Perform transformations to achieve efficient designs.* To efficiently implement the high level synthesis results, transformations at the hardware level such as data path partitioning and multiplexer reduction are extremely important. Therefore, one of the motivations on the hardware mapping is to recognize the important hardware transformations and to characterize the performances of those transformations.

- *Provide a framework to interface with various design styles and/or hardware modules.* The flexibility of choosing different clocking strategies, design styles, or cell libraries is achieved by the hardware database system. To provide the flexibility, the hardware database has the following characteristics: First, it contains all the important information which is necessary to the synthesis process. Second, the information in the database is ordered so that they can be easily accessed. Finally, the database is flexible enough to accommodate different clocking strategies and wiring options.

1.3 HYPER Overview

Before going into further details of the hardware platform, the overall HYPER synthesis system will be briefly described. This overview of HYPER is essential in understanding the design flow of HYPER and the interaction between the hardware platform and the other synthesis tools in HYPER.

HYPER consists of a library of software modules, operating on a centralized flowgraph database. An algorithm for real-time applications (described using Silage [60] [61] or entered in a schematic format) is first compiled into an intermediate *control data flow*

graph (CDFG), stored in the OCT database [23]. Silage is a signal-flow language developed especially for DSP specifications. It has many useful features such as fixed-point data types, sample delays, and multiple data rates which allow designers to quickly and precisely describe their DSP algorithms. Control macro constructs are being incorporated to enhance the expressive power of Silage. These macros include loops and if-then-else blocks to capture the global control flow of an algorithm. Figure 1.2 describes the well-known CORDIC algorithm [6] [73] [75] in the Silage language to show some of the features.

The CDFG is an one-to-one mapping from the Silage description. It is a hierarchical graph due to the control constructs: The body of a loop or a conditional statement is represented by a subgraph, which is compacted into a single node at the next hierarchy level. This representation has the advantages of compactness and expressiveness compared with a flat flow graph. More information about the CDFG can be found in [28]. After the parsing of the flow graph, a number of standard, architecture-independent, compiler transformations such as elimination of dead code, manifest expressions, common sub-expressions and algebraic identities, are executed.

Before synthesis is performed, simulation of the algorithm at the behavioral level is necessary not only to verify the functionality of the algorithm, but also to optimize certain algorithm parameters. A compiler has been developed to generate C codes from the CDFG description of an algorithm. The C codes take a command file and an input-sample file as inputs, and produce an output-sample file. The behavior simulation results can be used to cross check with the functional simulation results. The functional simulation will be performed on the structure generated by HYPER at the end of the synthesis process. This cross check provides a way to verify the correctness of the transformations performed over the synthesis steps.

The first step in the synthesis process is the *module selection* as described above. The synthesis process is then continued with operations such as estimation, allocation, transformation, assignment and scheduling. These operations are briefly described in this section. For details of each operation, please refer to [50] and [59].

In the estimation phase, min-bounds and max-bounds on the required resources are deduced. These bounds are important for several reasons: first of all, they delimit the design space, and thus speeding up the synthesis search process. Second, the computed min-bounds can serve as initial solutions for the allocation process. Finally, these bounds serve as parts of the resource utilization table to guide the transformation, assignment and

```
#define xyWord fix<22, 18>
#define phaseWord fix<22, 20>
#define N 20
#define Angle90 phaseWord (0.5)
#define Corrector xyWord (0.60725285)

func main(Xin, Yin: xyWord; CorAngles: phaseWord[N])
    Amplitude: xyWord; Phase: phaseWord =
begin
    (X[0], Y[0], Phi[0]) = if Yin >= 0 -> (Yin, -Xin, Angle90)
                           || (-Yin, Xin, -Angle90)
                           fi;
    (i : 1 .. N - 1) ::
        begin
            (X[i], Y[i], Phi[i]) =
                if Y[i - 1] >= 0 ->
                  (X[i-1]+(Y[i-1]>>(i-1)), Y[i-1]-(X[i-1]>>(i-1)), Phi[i-1]+CorAngles[i-1])
                  ||
                  (X[i-1]-(Y[i-y]>>(i-1)), Y[i-1]+(X[i-1]>>(i-1)), Phi[i-1]-CorAngles[i-1])
                fi;
        end;
    Amplitude = xyWord (X[N - 1] * Corrector);
    Phase = Phi[N - 1];
end;
```

Figure 1.2: CORDIC Algorithm in Silage

scheduling operations.

In the allocation phase, the number of execution units, the size of register files, and the distribution of the available time over subgraphs are decided either by human designers or by an initial guess from the estimation step. The goal of this phase is to come up with the minimal hardware configuration that meets the performance constraints.

The scheduling process assigns operations to so called control steps. The goal is to schedule all the operations within a given amount of time on the predefined allocation. Assignment determines on which particular execution unit a given operation will be realized, from which register the execution unit will request data, where the execution unit will send the result and which connection will be used for sending the data.

To ensure that the potential of a given algorithm is maximized, the application of optimizing transformations is a must. Transformations are changes in the signal flow graphs which improve the final implementation without altering the input-output relationships. Currently, several transformations including loop unrolling, constant multiplication expansion, and retiming for scheduling [48] have been implemented in the HYPER environment.

Allocation, transformation, assignment and scheduling are all interdependent processes. Designers can try out different designs of the same algorithm by modifying allocations and/or timing constraints to meet the required performance. The scheduling and assignment may fail in the synthesis process due to the fact that not enough resources are allocated. Under this circumstance, the design can either increase resources or invoke transformations to improve the resource utilization. Even when the scheduling and assignment succeed, the designer may still want to cut down resources to further improve the area efficiency. Therefore, the synthesis process is indeed a recursive procedure. Xhyper, a graphic front end of HYPER running on the X11 window system, has therefore been developed to help designers iteratively fire up these tools and find the optimal design. More details on the operation of xhyper can be found in [28].

After the synthesis process, the flow graph with the hardware assignment and scheduling information (called the decorated flow graph) is mapped into a hardware structure and silicon is generated using the Lager IV [20] silicon compiler.

HYPER was developed in a bottom up fashion. The hardware mapper was first developed based on the assumption that all the scheduling and assignment information is available at the step of hardware mapping. Then the scheduler was developed assuming

that the hardware allocation and selection have been completed. After developing the scheduler, we then started working on the hardware selection and the hardware allocation problems. Estimation was developed along with the scheduler. A set of transformations were incorporated into HYPER after the basic HYPER structure had been established.

This dissertation is organized according to the chronical order of the tools developed, emphasizing on the hardware platform of HYPER. In Chapter 2, the hardware mapping process including the problems encountered in this process and the solution we chose are presented in detail. In Chapter 3, the problems and techniques used in hardware selection are addressed. Chapter 4 demonstrates several benchmarks that have been successfully generated by HYPER. Finally, conclusions of this work along with the possible future directions are given in Chapter 5.

Chapter 2

Hardware Mapping

2.1 Problem Definition

Hardware mapping is the last step in the synthesis process, which consists of the mapping of the scheduled and allocated flow graph (or the decorated flow graph) into the available hardware blocks. Currently, the result of this step is a structural description in the sdl-language [20], which serves as the input to the Lager IV silicon assembly environment. The hardware mapper can also be adapted to different silicon assembly systems by generating different structural descriptions. The mapping process transforms the decorated flow graph into three structural subgraphs: the data path structure graph, the controller state-machine graph, and the interface graph. The interface graph determines the relationship between the data path control inputs and the controller output signals. This graph is important because it defines the overall clocking strategy of the data paths. Three dedicated mapping tools then translate these structural subgraphs into the corresponding structural views.

In addition to the algorithmic processes such as data path partitioning, multiplexer reduction and register-file merging, the hardware mapping process takes on several translation steps, which require an accurate knowledge of the available cell library in terms of functionality, speed, area, and black box view. This is provided by the rule-based library database, which is also a part of the proposed hardware platform and will be discussed later in this chapter.

For the control path, the hardware mapper first generates the state transition diagram from the scheduling information. Interface logic is then generated based on a

demand-driven approach. A finite-state machine is finally produced for the central control. Several control optimizations are performed in this process. One optimization example is the recognition of the control signals that are independent of control states and replacing them by a local control in the interface logic. The details of all these optimization steps are also discussed later in this Chapter.

This Chapter begins with a review of the input format and the target architecture of the hardware mapper, followed by a description of the related research in hardware mapping. After the description of the hardware database system, which is given in Section 5, the data path generation and control path generation is discussed in detail. Finally, this Chapter is concluded after a discussion of the order of the transformations.

2.2 Input Format

The input format to the hardware mapper is a decorated CDFG (Control Data Flow Graph). A simple example of this format is shown in Figure 2.1 which is part of the CORDIC algorithm. A detail description of the decorated CDFG can be found in Appendix C. The rest of this section briefly summarizes its characteristics.

2.2.1 Control Data Flow Graph

In a CDFG, a node can represent either an operation (called a *leaf node*) or an instance of a complete subgraph (called a *hierarchical node*). In this way, hierarchical graphs can be defined. Currently, HYPER supports two types of hierarchical nodes including *func* nodes and control macros such as *loop* (iteration) nodes and *if-then-else* nodes. Two types of edges are allowed in the CDFG. As shown in Figure 2.1, the solid edges represent the data dependency between two nodes and the dotted edges represent the control precedences between them. Only the solid edges are used in the hardware mapper to trace the interconnect between hardware modules. Control edges are ignored.

In Figure 2.1, two levels of hierarchies are shown. In the first level, three macro nodes represent the control flow of the algorithm. In the subgraph of each macro node (i.e. the second hierarchy), a data-flow graph is given to represent the data-flow operations inside the macro. The first func node describes the initialization phase of this algorithm. Inside the subgraph, some storage elements are reset or initialized. The loop (iteration) node represents the main loop of the algorithm. A data-flow description is given in the

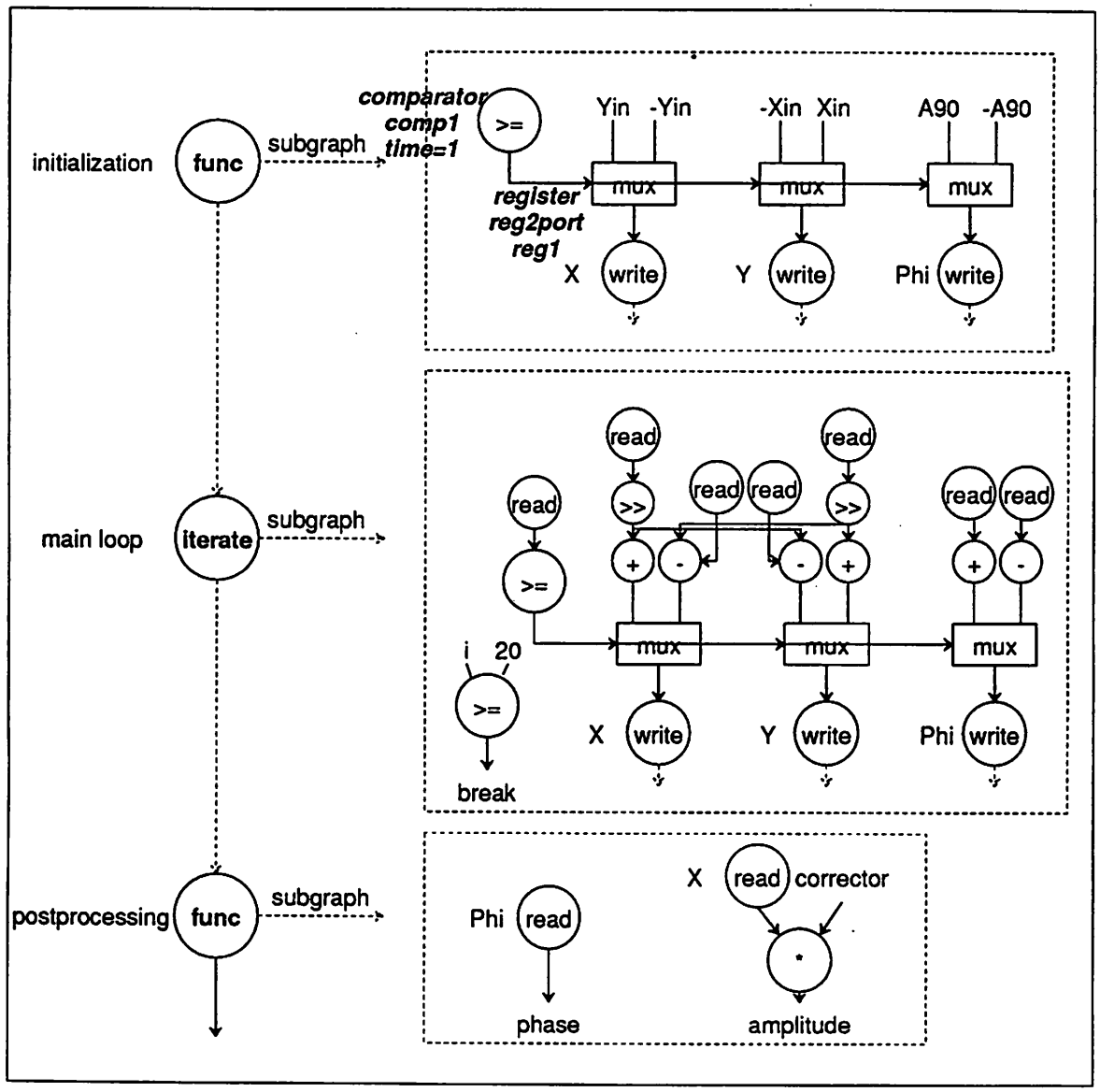

Figure 2.1: Flow Graph of the CORDIC Algorithm

subgraph to represent one iteration of the CORDIC algorithm. In addition to the data-flow operations, some operations which are necessary to test the ending condition of the loop are also included in the subgraph. These operations have to be performed after each iteration to decide whether to continue the next iteration. The last func node represents the postprocessing after the main loop. Final results of the main loop are corrected by a constant factor and sent off-chip. These operations are represented by the multiply node and the read nodes inside the subgraph of the last func node.

The reason for choosing the CDFG format is due to the fact that an efficient implementation of a high-performance system requires an unambiguous description of the overall control flow of the system as well as the data-flow description. Purely control-flow description cannot represent the parallelism, which is essential in high-performance real-time systems. Purely data-flow description, on the other hand, does not have the control macro constructs or the control precedences, which are important for an efficient controller synthesis. Therefore, a CDFG format is essential in synthesizing real-time algorithms.

Hierarchy is another important feature of the representation. Hierarchy helps in keeping the original construct of an algorithm specified by the designers. The original construct usually includes structural hints from the designers, and therefore is very important for the synthesis process. Hierarchy is also very important for reducing synthesis time. Without the hierarchy of the iteration nodes, the large amount of vertices and edges of the flat graphs will greatly slow down most of the synthesis tools.

2.2.2 Decorated CDFG

The result of the synthesis operations, including hardware selection, allocation, assignment, and scheduling, is back-annotated onto the flow graph. The hardware mapper then translates the *decorated* flow graph into a hardware structure. For each operation node, the required decoration includes a positive integer to represent the relative order of the operation and the type and the name of the operator on which the operation will be performed. For a hierarchy node, only the scheduling information is needed. For special nodes such as merge nodes or bit-select nodes, no decoration is required by the mapper. For each edge, information on the storage type (register of variable), the name of the storage element, and the implementation of the storage element is required for the mapping process. Figure 2.1 shows the decoration of one comparison node and the decoration of the output

edge of that node as an example of the decoration.

2.3 Target Architecture

A real-time system is normally a heterogeneous composition of architectures and components. These architectures can be classified into several categories based on the amount of operation sharing on an arithmetic unit. One extreme end of the scale represents the traditional micro-processor architecture, where all operations are time-multiplexed on one single general-purpose ALU. This architecture is classified as *control driven.* On the other end of the spectrum, one can find architectures such as systolic arrays, where each operation is represented by a separate hardware unit. This architecture is called *hard-wired* or *data-flow driven.*

In the computation intensive parts of the real-time systems, the data rate often approaches the maximally achievable clock rate. In those cases, the use of a cluster of dedicated bit-sliced data paths with extensive pipelining and limited resource sharing is unavoidable. Examples of such architectures are found in speech recognition [27] and image processing systems [63], where the data rates are at the order of 10M samples per second. The *bit-sliced data path cluster* is exactly the target architecture of HYPER. The characteristics of this kind of architecture are that the data paths are hard wired in order to match the algorithmic data flow. The amount of programmability is very restricted. The controller section for those architectures is therefore small compared to the data path and memory blocks, but has to be fast and efficient.

Figure 2.2 shows the template of the HYPER target architecture. This template involves a data path cluster, a Finite-State Machine (FSM), a set of control slices, and other modules such as array multipliers and on-chip memories. The data path cluster includes modules such as the execution units, registers, and multiplexers. These modules are partitioned properly to achieve layout efficiency. An on-chip Finite-State Machine (FSM) is generated by the mapper as the central control. Between the central control and each data path is a control slice for the interface logic and the local control. Control signals may also go directly from the FSM to the data paths or vice versa without going through the interface logic. Other modules such as on-chip cache memories, array multipliers, or register files are also possible in this architecture to achieve a higher processing rate. There are two restrictions on this architecture template. First, only one central control is allowed.

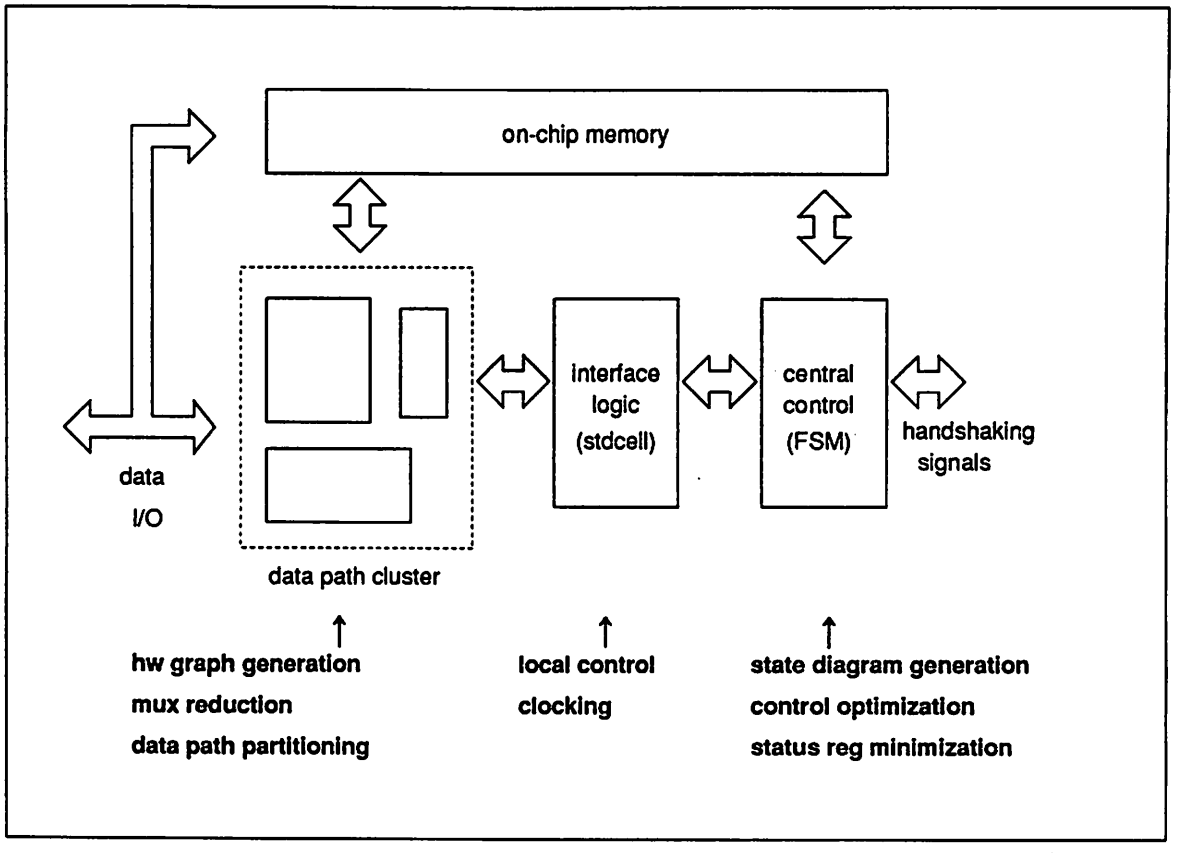

Figure 2.2: Target Architecture of HYPER

Second, the design must be synchronous. Even with these restrictions, this architecture is powerful enough to implement the applications that HYPER is targeted at.

Figure 2.3 shows a simple example of the mapping process. A decorated flow graph produced by the synthesis process is taken as the input to the hardware mapper and a register transfer level description of the decorated flow graph is generated by the mapper.

2.3.1 Clocking Strategy

Currently, the hardware mapper assumes a two-phase non-overlapped clock in the target architecture. Registers are loaded at one of the phases (e.g. phase one) and register outputs are enabled at the other phase (e.g. phase two). The same assumption also applies to the memories and the state registers of the FSM. Other clocking strategies can also be implemented by HYPER since the clocking information of each module is specified by

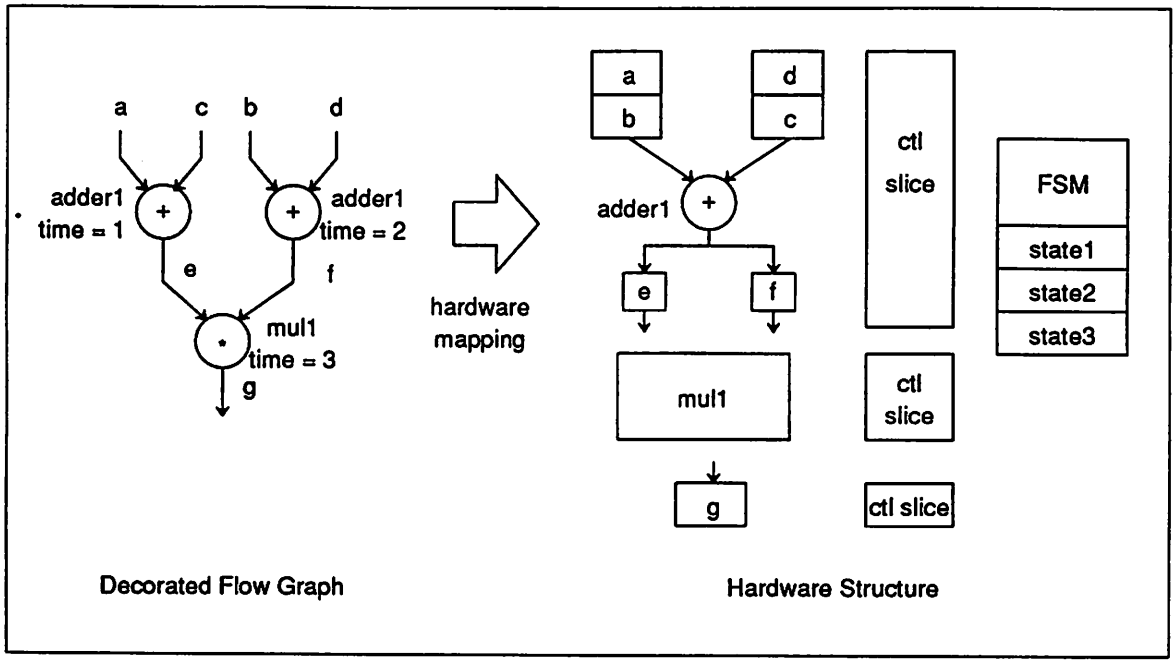

Figure 2.3: Example to Illustrate Hardware Mapping

the designer in the hardware database library. The CTL-IN-TERMINAL attributes in the database provide the values of the control inputs, which also specify a proper phase of the clock. With the modification of the CTL-IN-TERMINAL attributes and some simple changes in the mapper, synchronous designs of other clocking strategies can be synthesized.

Notice that all the modules in the same database library should have a consistent clocking strategy to ensure the correctness of synthesis results. For example, assume that Register A and Register B are two types of registers in the same database. If Register A loads data at phase one and outputs data at phase two, Register B should also loads data at phase one and outputs data at phase two.

2.4 Previous Work on Hardware Mapping

Not many synthesis systems address the hardware mapping issue. Even systems that address hardware mapping have different approaches due to different architectures and optimization goals. Cathedral III [39] emphasizes on the timing optimization of execution units. The synthesis process of Cathedral III starts with a one-to-one mapping of the

function nodes to the execution units. The system then tries to optimize the timing of the execution units so that the units can be multiplexed. In HYPER, the optimization goal of the hardware mapper is on the area efficiency. The execution units are selected from a hardware cell library in a previous module selection phase based on the resource utilization and the hardware cost. With the execution units selected and scheduled, the hardware mapper simply performs transformations to achieve efficient area utilization.

The AT&T Mind [21] system aims toward the logic optimization. Comparing with Mind, the emphasis of HYPER is on the data-path optimization, even though HYPER performs logic optimizations as well. The logic optimization in HYPER can be performed by both the control-path optimization in the hardware mapper and by the MIS tool [34], [43], which is invoked in the layout generation phase. Another difference between the hardware mapper of HYPER and Mind is the layout style. Mind chooses the standard cell implementation, while the HYPER hardware mapper focuses on the bit-sliced data path structure. Therefore, some issues such as data path partitioning are not considered in Mind, but are extremely important for the HYPER mapper.

Similar to the Mind system, Register Transfer Level (RTL) synthesis systems such as the VHDL Synthesis System (VSS) of University of California at Irvine [56] and the Synopsys tools also aim toward the logic optimization. In addition, resource allocation, state assignment and test vector generation are emphasized. Tasks such as register grouping and data path partitioning, which are automatically performed by the HYPER hardware mapper, are performed manually by human designers in those systems.

The CMU's LASSIE [70] system provides a flexible framework for mapping the structural output of the System Architect's Workbench [22] onto a variety of layout styles. The hardware mapper in HYPER is not as general, but can be extended to provide similar features. Compared with LASSIE, the HYPER mapper is more concentrated on the performance of the transformations rather than providing a general mapping environment.

Feedback-driven data-path optimization is proposed in Faslot [14] to improve layout efficiency. In HYPER, this process can be performed interactively between the designer and the hardware mapper. Since the layout generation normally takes hours to perform, it is impractical for the hardware mapper to automate this iterative process. Instead, the mapper provides useful data such as the estimated data path area and the number of nets for each module to help the designer in choosing proper hardware implementations before performing the layout generation.

The emphasis of the HIS system [33] from IBM is on the allocation of the hardware resources including functional units, registers, and buses. The optimization is based on path analysis and mutual exclusiveness. Many technology dependent optimizations that HYPER performs are not considered in HIS since no real hardware platform is assumed.

The key observation from comparing various hardware mappers is that hardware mapping should be built on top of the hardware models and the optimization goals. Since HYPER is aimed at high-performance real-time applications, the proper architecture template is the bit-sliced data path cluster. Based on the well-defined architecture template, the HYPER mapper can perform many useful data path optimizations to improve the hardware efficiency.

2.5 Hardware Database System

Before describing the hardware mapper in further detail, I will first describe the hardware database system. The database system contains information such as delay, area, hardware parameters, and black-box views of the hardware blocks, which constructs the register-transfer level hardware structure. Currently, the database is implemented in the Lisp syntax. It can be rewritten in the OCT format when the structure of the database is finalized. A detailed description and a user's manual of the database system can be found in Appendix B. This section will highlight the major features and the design philosophy of the system.

2.5.1 Database Format

Hardware blocks in the Lager silicon compilation system [20] can be divided into three categories: data-path blocks, array blocks, and standard-cell blocks. The HYPER database system is organized accordingly. *Rb-dp* addresses the data-path blocks such as adders, registers, and multiplexers. *Rb-array* focuses on the array blocks including memory blocks, Programmable Logic Arrays (PLA's), and execution blocks such as multipliers. *Rb-std* is designed for the standard cells, which are mainly used in the interface logic. The basic structures of the three databases are the same, but the detailed information for each category is different due to the applicability of the information to these blocks. The basic structure of the databases can be described in the following format:

```
database :=
    ((function1
        (cell-name1
                (item-name1 data1)
                (item-name2 data2)
                ;; other items of cell 1
                )
        (cell-name2
                ;; items of cell 2)
        ;; other cells of function1)
     (function2
        ;; cells of function2)
     ;; other functions)
```

From this format, we can see that the basic structure is a hierarchical database with function names as the primary key. Under a function name, several blocks that implement the function are provided. Multi-function blocks will appear under all the functions that they can perform. For each block, information such as parameters, area and delay is specified. This information is called the attributes of the hardware block. For multi-function blocks, some attributes such as delay and parameters are function dependent and therefore may have different values for different functions.

In addition to the above structure, the database can also be organized in the following format with the cell name as the primary key and the function name as the secondary key:

```
database :=
    ((cell1
        (function1
                (item-name1 data1)
                (item-name2 data2)
                ;; other items of cell 1 performing function1
                )
        (function2
                ;; items of cell 1 performing function2)
```

```
    ;; other functions that cell1 performs)
(cell2
    ;; functions of cell2)
;; other cells)
```

The HYPER hardware database supports both structures and provides access routines so that the user can use either the function name or the cell name as the primary key to access the database.

As described above, the three databases may have different attributes. The reason is that some attributes are only applicable to a certain types of blocks. For example, modules in rb-dp have a parameter attribute, which specifies the hardware parameters such as the bit width of the modules. This parameter attribute doesn't apply to the modules in the standard cell database since this database is designed for the glue logic and the bit widths of these modules are always one. Different databases can have the same attribute, but the value of the attribute may have different formats. For example, the parameter attribute of the data path modules is usually a single parameter, namely the word width. For the array modules, however, the parameter attribute usually involves a list of parameters such as the input width, the output width, and the array contents of the modules. Therefore, different attribute formats are defined for each database.

2.5.2 Accessing the Database

The search of the database can be driven either by timing or by area constraints. All the cells are stored in the database in the increasing order of the cell size so that the search can be performed efficiently. However, the *cheapest-cell-first* rule doesn't apply if the block size can not be evaluated before the silicon compilation process. The structure of certain hardware blocks such as Programmable Logic Arrays (PLA's) depends on the logic synthesis tools, and therefore the block area can't be evaluated until the layout generation phase. Those types of blocks will be stored in the database according to the *newest-cell-first* order[1]. A set of access routines have been implemented to facilitate the interface between the HYPER tools and the databases. In addition to the access routines, a database editor has also been implemented under the Lisp environment. This editor provides the cell designers with functions to insert, delete, or update a hardware block or the attributes of the block.

[1]That is, the latest generated module will be stored on top of the others so that it can be easily accessed.

2.5.3 Examples of the Database

Several examples will be shown in this section to illustrate the basic features of the hardware database. The complete hardware database which describes the Lager cell library can be found in Appendix B.

Example 1:

```
(+ ("adder" (PARAMETERS (N))
            (AREA (* N 48 214))
            (DELAY (+ 6 (* N 2.2)))
            (ONE-BIT-DELAY 6)
            (RIPPLE-DELAY (* N 2.2))
            (RIPPLE-OFFSET 0)
            (DATA-TERMINAL (OUT (IN1 IN2)))
            (POWER-TERMINAL (Vdd GND))
            (CTL-IN-TERMINAL ((CIN GND) (CININV VDD)))
            (CTL-OUT-TERMINAL COUT)
            (CTL-TERM-EDGE ((CIN BOTTOM) (CININV BOTTOM)
                           (COUT TOP)))
            (COMPLEMENT-OUT OUTINV)
            (DRIVING-CAP NO))
   ("fastAdder" (PARAMETERS (N))
                (AREA (* N 48 250))
                (DELAY (+ 4 (* N 1.2)))
                ....))
```

This example demonstrates the structure of the database. In this example, + (addition) is the function name and the primary key to access the database. Under the key, two modules, "adder" and "fastAdder", are described. Notice that "fastAdder" has a shorter delay and a larger area; therefore, it is listed after the "adder" block according to the *cheapest-cell-first* rule.

This example also demonstrates the basic attributes of a data-path block. The first attribute of a data-path block is a list of hardware parameters of the block. In this example, "adder" has only one hardware parameter, N, which represents the block width. The second and the third attributes are the area and the delay of the block. Both attributes are Lisp expressions of the hardware parameters. The ripple delay, one-bit delay, and ripple offset attributes are used for the ripple model in the hardware selection phase. These attributes are dealt with further in Chapter 3.

The data-terminal attribute specifies the output terminal and the input terminals of the block. Each block can have only one output terminal for a specific function, while its input-terminal attribute can be a list of terminals. If a module has more than one output terminal, the data-terminal attribute should specify the output terminal that carries the result of the given function (called the *primary output terminal*). The complementary output can be specified in the complement-out attribute and the other outputs are ignored in this description since these outputs are carrying values of other functions.

The power-terminal attribute specifies the power terminals that need to be connected, either to power (Vdd) or to ground (GND). The control input/output terminal attributes not only specify the terminal names, but also the values of these terminals for that specific function. This information will be used to derive the interface logic of the control path. The control-terminal-edge attribute specifies which side a control terminal should be brought out. The hardware mapper can make use of this information to improve the layout efficiency. The driving-capability attribute specifies whether the hardware module has an output buffer allocated inside the module. This information is important for the hardware mapper to allocate extra buffers if there are many fan-outs in the design.

Example 2:

```
(<< ("shift" (PARAMETERS (N (SBY (STRUCTURE M))))
             (RANGE (-31 31))
             (AREA (* N 50 (+
                    (* (myLength SBY (list "up1" "down1")) 84)
                    (* (myLength SBY (list "up2" "down2")) 90)
                    (* (myLength SBY (list "up4" "down4")) 102)
```

```
        (* (myLength SBY (list "up8" "down8")) 130)
        (* (myLength SBY (list "up16" "down16"))
         186))))
(DELAY (* 1.5 (length SBY)))
(ONE-BIT-DELAY (* 1.5 (length SBY)))
(RIPPLE-DELAY 0)
(RIPPLE-OFFSET M)
(DATA-TERMINAL (OUT (IN)))
(POWER-TERMINAL (Vdd GND))
(CTL-IN-TERMINAL
 ((shift BUS ((length SBY)
              (encoded_SHIFT SBY)))
  (shiftbar BUS ((length SBY)
                 (NOT (encoded_SHIFT SBY))))
  (msbin BUS ((length SBY)
              SHIFT_IN_NUM))))
(CTL-OUT-TERMINAL NIL)
(CTL-TERM-EDGE
 ((shift TOP) (shiftbar TOP) (msbin TOP)))
(COMPLEMENT-OUT NIL)
(DRIVING-CAP NO)))
```

This example illustrates a data-path block with control buses and multiple parameters. "Shift" is a log shifter which composes various numbers of sub-blocks to perform the left shift operation (<<). The shift range is specified by the range attribute. A negative number in the shift range represents shifting toward the opposite direction. In this example, the maximal number of bits that "shift" can perform is 31 bits (either to the right or to the left). This log shifter has two hardware parameters: N is the block width of the shifter and SBY is the structure of shifter. SBY can be derived from the number of bits to be shifted (M). For example, M = 7, the log shifter should contain three sub-blocks: up1 (which shifts one bit toward the left), up2, and up4. Therefore,

```
SBY = STRUCTURE(M) = ("up1" "up2" "up4")
```

STRUCTURE is a function defined in the database to return a list of sub-blocks based on the integer M.

The area of the log shifter is a complex Lisp expression. A function "myLength", which calculates the number of a certain sub-block in the SBY list, is pre-defined in the database. The area expression is then specified using this function. Continuing from the previous example, M = 7 and SBY = ("up1" "up2" "up4"), we have:

```
(myLength SBY (list "up1". "down1")) = 1
(myLength SBY (list "up2" "down2")) = 1
(myLength SBY (list "up4" "down4")) = 1
(myLength SBY (list "up8" "down8")) = 0
(myLength SBY (list "up16" "down16")) = 0
```

Therefore, the area of this specific log shifter is:

```
N * 50 * (84 + 90 + 102)
```

Notice that *myLength* is different from *length*, which is a Lisp function and returns the number of elements in a given list. For example, in the expression for the one-bit-delay, we have:

```
(* 1.5 (length SBY)) = (* 1.5 3) = 4.5
```

To represent the control bus, a keyword "BUS" is given following the control terminal name. Both the value of the control bus and the width of the control bus have to be specified in the control-terminal attribute following the BUS keyword. The format for the control bus can be defined as follows:

```
control bus :=
   (terminal-name BUS (bus-width bus-value))
```

Both the bus value and the bus width can be Lisp expressions. In the log shifter example, the bus width is:

```
(length SBY) = 3
```

and the bus value is specified in terms of a pre-defined function "encode_SHIFT", which takes SBY as the input and returns encoded values for the control bus.

Example 3:

```
(RAM ("RAM3T" (PARAMETERS (("in-width" INWIDTH)
                           ("out-width" OUTWIDTH)
                           ("ram-address-plane" INPLANE)
                           ("ram-bit-plane" OUTPLANE)))
              (AREA (* "in-width" "out-width" 31 28))
              (DELAY (+ (* 2 "in-width") (* 2 "out-width")))
              (DATA-TERMINAL (RAMDATABUS (RAMADDRESS RAMDATABUS)))
              (POWER-TERMINAL (Vdd GND))
              (CTL-IN-TERMINAL ((PHI1 CK2) (PHI2 CK1)
                               (READ OEN) (WRITE LOAD)))
              (COMPLEMENT-OUT OUTINV)
              (DRIVING-CAP SMALL)
              (REDUCIBILITY NIL)))
```

This example shows a RAM module "RAM3T" in the array database. This module has multiple hardware parameters: in-width, out-width, ram-address-plane and ram-bit-plane. Each parameter is defined as an expression of a set of keywords, of which the definitions can be found in Appendix B. The expression can be any Lisp function or user-defined function such as "log" and "exp".

Unlike the data-terminal attribute of the data-path modules which allows only one output terminal specification, the data terminal attribute of memory modules is able to represent multiple-port cases. For this example, RAM3T has only one I/O port and therefore the representation is the same as the previous examples and can be simply given by an output terminal followed by an input terminal list (an ordered list). For multiple-port cases, the format of the attribute is a more complicated expression and can be found in Appendix B.

The reducibility attribute specifies if the output is tri-state buffered. If this attribute is t (true), the output of the memory module is tri-state buffered and therefore the memory is *multiplexer reducible*. Otherwise, this module is not multiplexer reducible.

Multiplexer reduction will be discussed later in this chapter. The other attributes of the array modules are similar to those of the data path modules.

This example assumes a two-phase non-overlapped clock, which is the clocking strategy of most Lager designs. Based on the clocking strategy, the values of the control input terminals are defined. This module can also be used in designs of other clocking strategies by modifying the value of the control-input-terminal attribute.

2.6 Data Path Generation

Having described the hardware database system, we now start the discussion of the hardware mapper with the data path generation. Data path generation is to produce a data path description in the sdl-language from a decorated flow graph. Figure 2.4 outlines the major steps in the data path generation. Allocation and assignment information of the decorated flow graph is needed for the hardware graph generation step and scheduling information is required for the register file merging step.

The data path generation process begins with the initialization step, which initializes a symbol table and some variables for the hardware mapper. The second step of the process is to interpret the assign operations (i.e. equal operations) of the decorated flow graphs. Assignments are used to represent data transfers and can be implemented in several ways. For instance:

- If a variable is set to zero and this variable is stored in a counter or a resetable register, this equal operation can be implemented by a reset of the counter or the register.
- If a value is assigned to more than one register or execution unit, this assignment is a broadcast operation.
- If both the input value and the output value of an equal operation are stored in registers, this operation represents a register transfer.

The hardware mapper has to find the cheapest way to implement the equal operations. A simple rule-based system is used in the hardware mapper to solve this problem. This rule-based system allocates proper hardware units and decides proper control operations for the equal operations by checking the attributes of the input and output edges of the assign nodes.

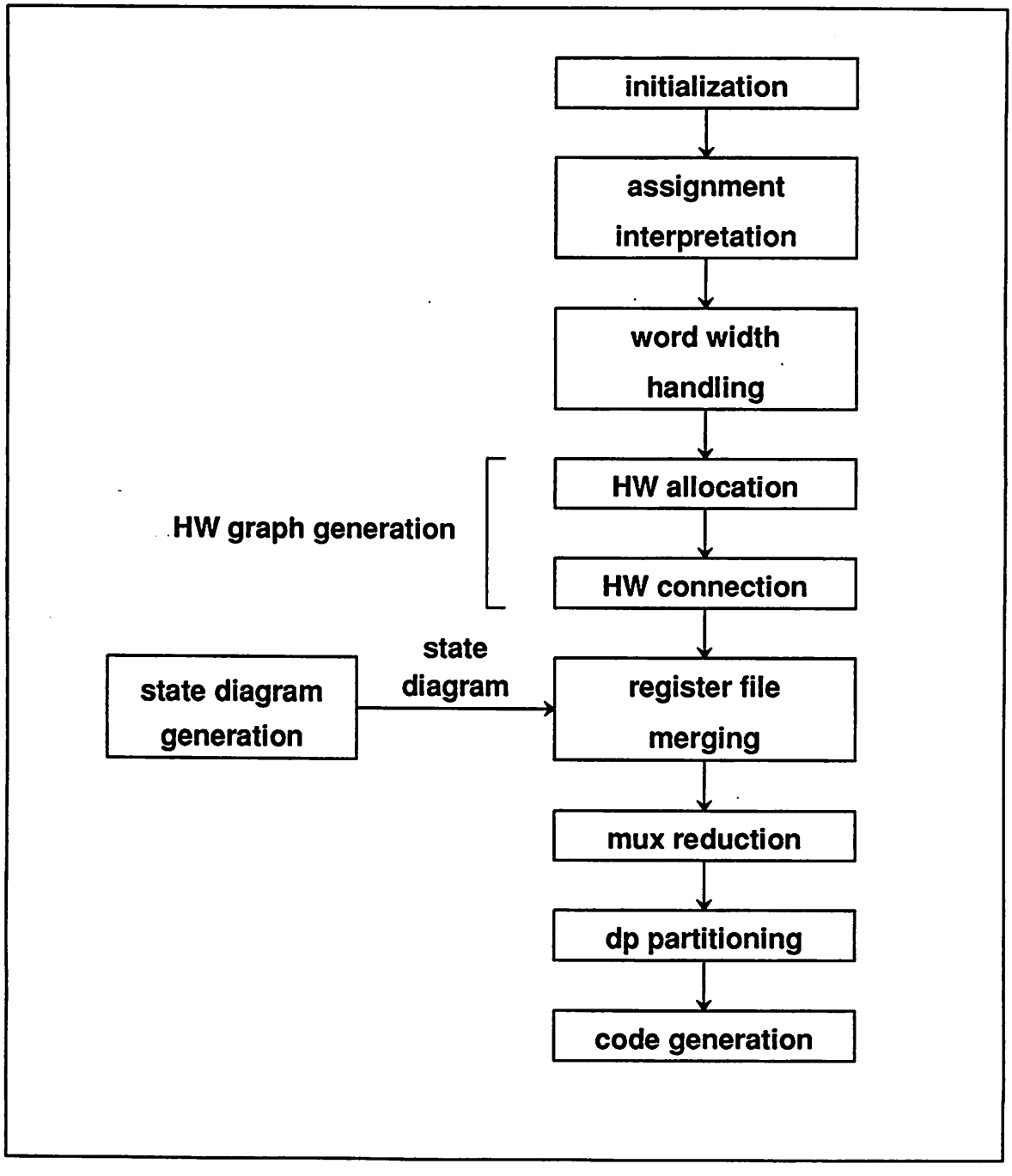

Figure 2.4: Major Steps of Data Path Generation

The third step in the data path generation involves deciding the bit width for each hardware module. Most of the nodes and the edges in the decorated flow graph have been given the width attribute before hardware mapping. However, some of them such as constant edges, which will be mapped to constant modules, may not have a proper width. The hardware mapper will assign the width attributes for those modules according to a set of rules. Those rules are:

- For control edges, which may be mapped to control registers, their widths are 1.
- Derive the width of a node from its input/output edges and derive the width of an edge from its input/output nodes.
- A constant edge can have a width equal to $\lceil log_2(value) \rceil$.
- Else take the default value, that is, 1.

The Cathedral System [24] also addressed the data alignment problem. The focus is on the hardware minimization (including multiplexers and routing) and the approach is to formulate the problem into an integer linear programming problem. Compared with the Cathedral System, HYPER uses a simpler rule-based approach. The reason is that most of the data alignment problems can be solved by a set of derivation rules in the Silage compiler. For problems that have not been solved by the derivation rules, the hardware mapper simply applies the rules described above to complete the synthesis process. Although the rules are easy, they provide a feasible solution without losing much hardware.

After the three preprocess steps as described above, the hardware mapper is ready to generate hardware by putting down hardware blocks and connecting them. The result of this step is a *hardware graph* in which each node represents a hardware block and each edge represents a net. Hardware blocks include registers, constant blocks, and execution units such as adders and shifters. To connect the hardware blocks, multiplexers or tri-state buffers may be needed for nets with multiple fan-in's. However, some of the multiplexers can be reduced later in the multiplexer reduction step. The resulting hardware graph is a flat graph, unlike the flow graph which is hierarchical. A hierarchical hardware graph will be built later when registers are merged into register files.

After the hardware graph is generated, several optimizations can be performed. The major optimizations implemented in the hardware mapper include register file merging, multiplexer reduction, and data path partitioning. These three optimizations are further

discussed in the following subsections. Notice that these optimizations will not change the functionality of the hardware graph. Their purpose is to improve the area efficiency. The final step of the data path generation is code generation which produces the data path sdl-files.

2.6.1 Register File Merging

1. Motivation and Problem Definition

The purpose of register file merging is to improve area efficiency. By merging registers into register files, all the registers in a register file can share the same data bus and use a decoder for the control. In addition to improving area efficiency, register file merging also reduces the problem size for the data path partitioning since a register file can be treated as a single module instead of several entities. Figure 2.5 illustrates the advantages of the register file merging. In this Figure, Registers a, b, c, and d are merged into a register file. Both the number of the input buses and the number of the output buses reduce from 4 to 1. The number of control lines reduces from 4 to 2 with a local decoder allocated.

There are two constraints for the register file merging: All the registers in a register file have to be tri-stated and cannot have any I/O conflicts since all the registers share the same data bus. By having no I/O conflicts, I mean that the registers can not load data at the same time, nor can they enable outputs simultaneously.

Register file merging may affect the number of multiplexers (or tri-state buffers) needed. Figure 2.6 shows two examples where the number of the tri-state buffers increases and decreases respectively after the merging. Assume that all the registers have tri-stated outputs. In Example 1, the number of the tri-state buffers increases from 0 to 1. In Example 2, the number of the tri-state buffers decreases from 2 to 1. The rules for allocating tri-state buffers are explained later in this section.

From the above description, the problem of the register file merging can be defined as follows:

Register file merging is to merge those registers with tri-stated outputs into register files so that there is no I/O conflicts and the cost function, which includes the total number of the data buses and the number of the multiplexers required, is minimized.

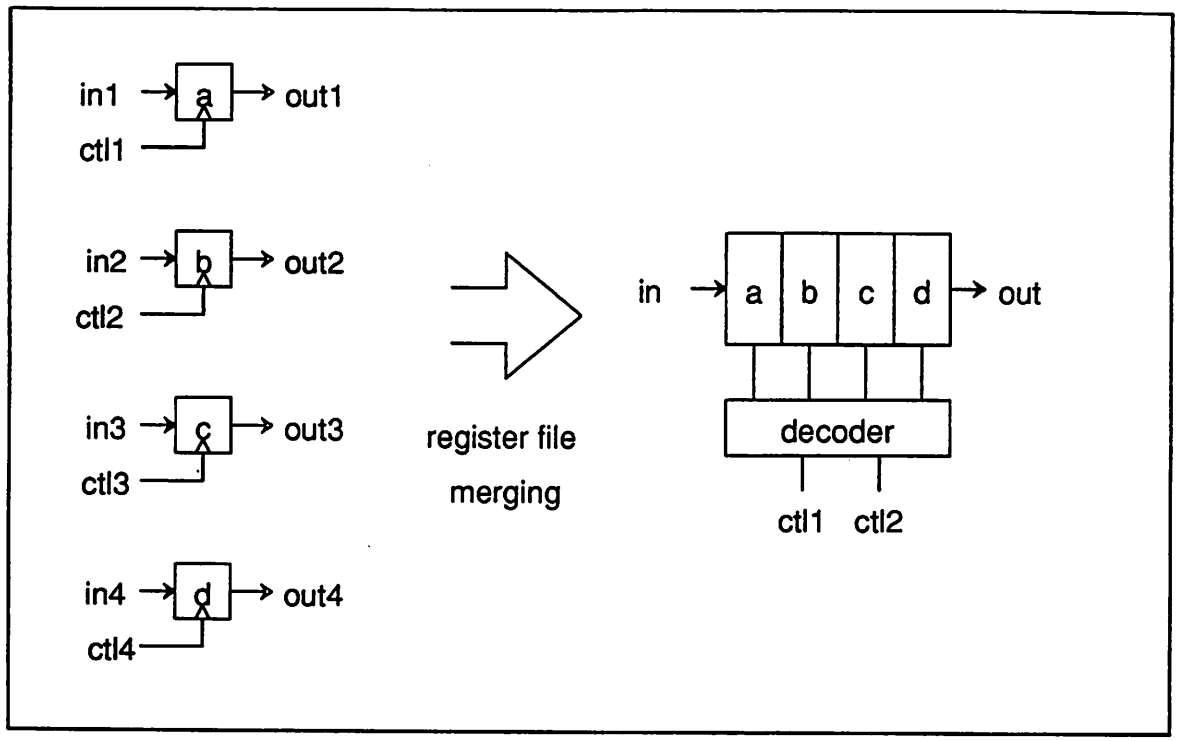

Figure 2.5: Register File Merging

If the cost of the multiplexers is not considered and the registers in a register file are treated as the nodes of the same color, the register file merging problem is exactly the graph coloring problem [42], which is NP hard. Simulated annealing [38] is known as a good approach for solving NP-hard problems such as placement problems, wiring problems [38] [41] and resource allocation problems [15] and is therefore chosen to solve the register file merging problem.

Even though the register file structure is part of the hardware model in the allocation and scheduling phase of HYPER as shown in Figure 2.7, the hardware mapper still tries to solve the register file merging problem for two reasons. First, the hardware mapper is designed to be general enough to handle any hardware structure and therefore no fixed hardware model is built in. Even without the register file information from the prior synthesis process, the mapper should be able to merge registers to reduce the hardware cost. Second, the scheduler and allocator do not take the multiplexer cost into account and therefore the register file structure from the allocator may not be suitable at this level of abstraction. Table 2.1 shows some examples in which the number of tri-state buffers reduces

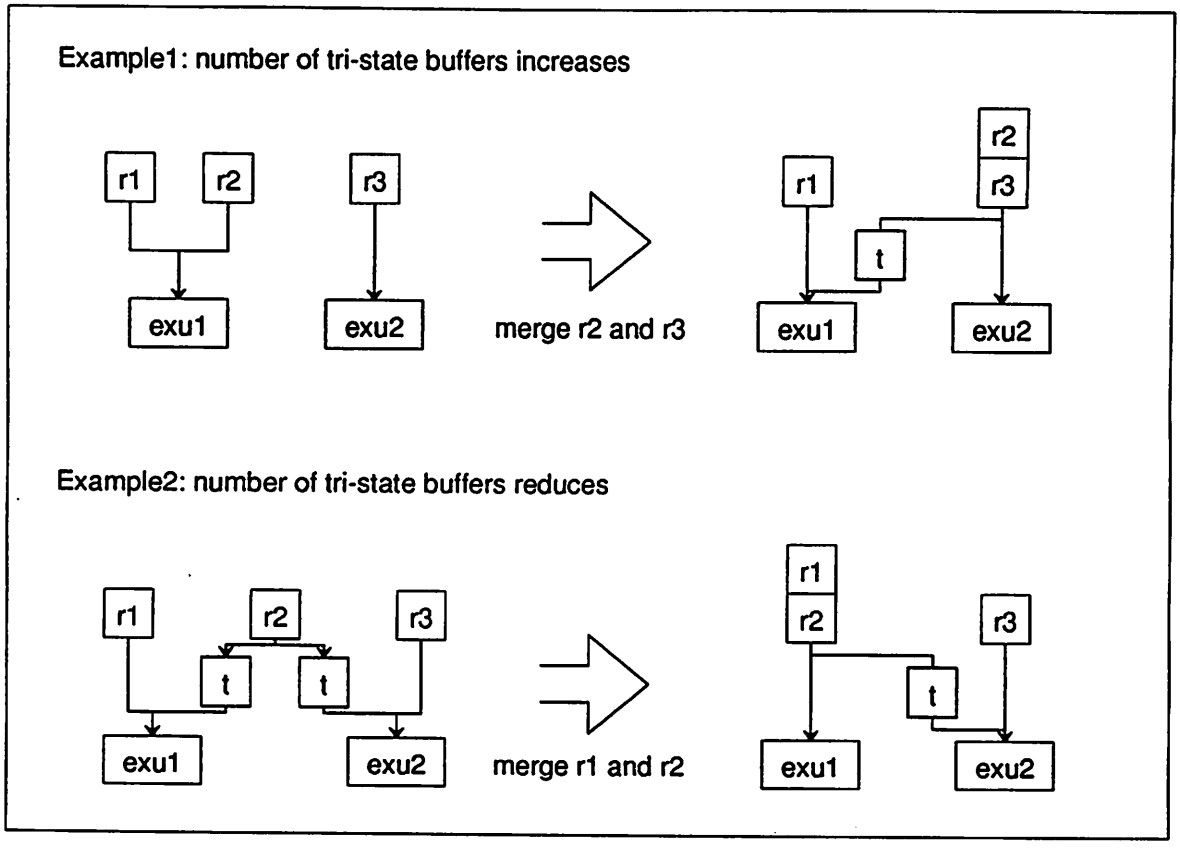

Figure 2.6: Multiplexer Increasing/Decreasing Due to Register Fi le Merging

after the mapping process. The numbers of tri-state buffers for three different structures are shown in this table. The first column represents the register file structure from the allocator, the second column represents the register file structure after the mapping process, and the third column represents the structure in which each register is treated individually (that is, no register is merged.). We can see from this table that reduction in the tri-state buffers is achieved for most examples by performing the register file merging.

2. Simulated Annealing Approach

To solve the register file merging problem, the following information has to be provided: the interconnect of all the hardware blocks, the information whether a hardware block is tri-state buffered, and a constraint graph representing the I/O conflicts among the registers. This information can be obtained as follows: the interconnect is found from the hardware graph, the tri-state buffered information is obtained from the hardware database,

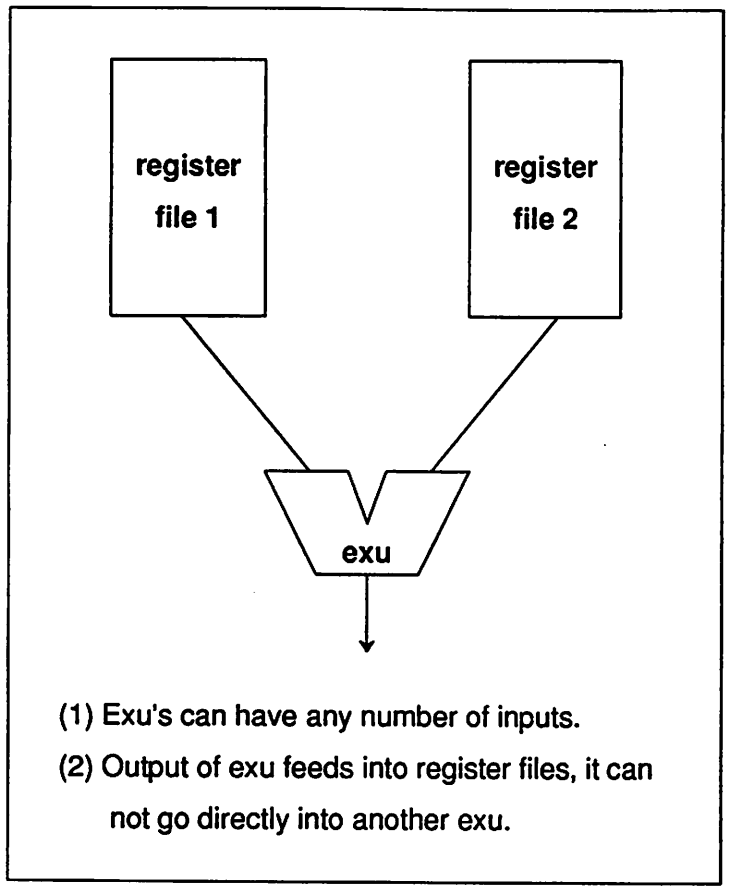

Figure 2.7: Hardware Model of HYPER

and the constraint graph can be derived from the state transition diagram, which is produced in the control path generation process from the scheduling information of the flow graph. The state diagram generation is discussed in the control path generation section of this chapter.

With the required information, the register file merging problem can now be re-formulated as follows: Given the interconnect of the hardware blocks and a constraint graph representing the register I/O conflicts, find a new hardware structure with tri-stated registers merged into register files so that the cost function given below is minimized.

$$Cost = number\_of\_register\_files + C * cost\_of\_multiplexers$$

Where C is a constant weighting factor. Notice that if C is equal to zero, the register file merging problem reduces to a graph coloring problem. However, the second term is usually

Examples	Before Mapping	After Mapping	No Register File
IIR, v1	40	28	31
IIR, v2	23	18	19
IIR, v3	19	17	19
FIR	5	5	5
Volterra	8	7	8

Table 2.1: Number of Tri-state Buffers Needed Before and After Mapping.

more important than the first term in the merging problem and therefore, the register file merging problem cannot be treated as the graph coloring problem. From some experiments, C should be set to around 10 to find good solutions.

The number of register files can be easily calculated; however, to find out the multiplexer cost, a more complicated rule has to be followed. Before giving the rule, the unit of the multiplexer cost is defined. Since all the multiplexers can be implemented by tri-state buffers and each tri-state buffer represents one input of a multiplexer, we can simply use the number of the tri-state buffers needed (i.e. the number of the total multiplexer inputs) as the multiplexer cost. In the Lager cell library, the cost of a 2-to-1 multiplexer is almost twice the cost of a tri-state buffer[2]. Therefore, using the number of multiplexer inputs as the multiplexer cost is an easy and reasonable approach to calculate the cost. For cell libraries with different costs of tri-state buffers and multiplexers from those of the Lager cell library, using tri-state buffers to implement multiplexers may not be desirable since different implementations of multiplexers may lead to a lower cost. In this case, the multiplexer costs cannot be defined in terms of the number of the total multiplexer inputs. A look-up table which includes the costs of various multiplexers is needed for this case to calculate the multiplexer costs. In the next section, a simple algorithm to build this table from the hardware cell library is described.

To calculate the multiplexer costs, we need to know when a multiplexer is needed. Consider an input terminal of a hardware module: if the terminal has only one fan-in, no multiplexer is required. On the other hand, if the terminal has more than one fan-in (say N), a N-input multiplexer is needed. However, there are cases where the multiplexers can be removed or *reduced*. A module is said to be *multiplexer reducible* if its output is tri-state

[2]In the Lager cell library, the delay values for the tri-state buffer and the 2-to-1 multiplexer are 1.2 nsec and 2.2 nsec respectively. The areas for these two cells are 1900 λ^2 and 3588 λ^2. The area ratio is 1.89.

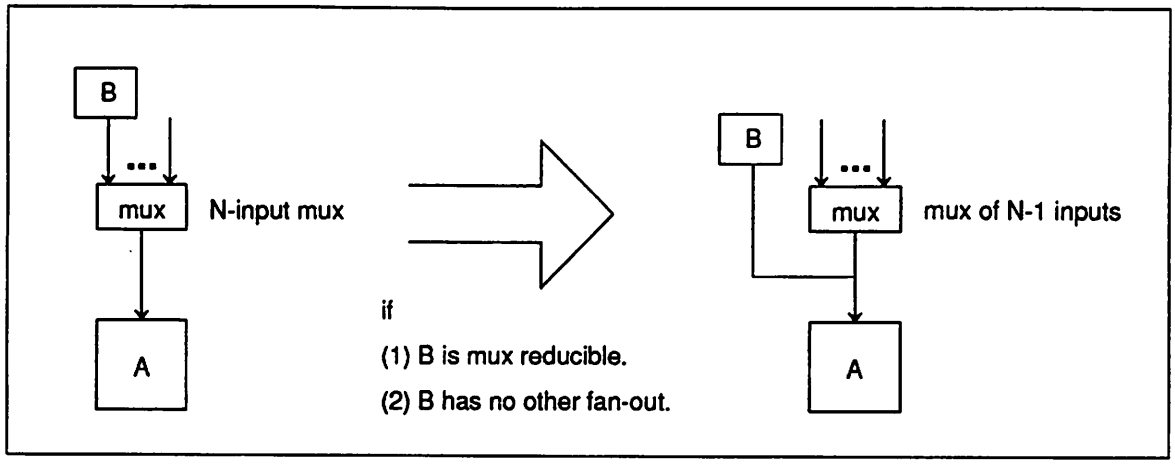

Figure 2.8: Multiplexer Reducible Modules

buffered. Examples of such modules are tri-stated registers, register files, and memories. The multiplexer reducibility is one of the attributes in the hardware database library and will be examined for each module in the hardware graph. If a multiplexer-reducible module is one of the N fan-in's and this module doesn't have any other fan-out, it can be removed from the inputs of the multiplexer and directly connected to the output of the multiplexer. The N-input multiplexer is now reduced to a multiplexer of $N-1$ inputs. Figure 2.8 illustrates the idea. The no-other-fan-out condition as described above of the multiplexer-reducible module of Figure 2.8 is needed to guarantee that this module only enables its output when Module A requires the output.

We can now describe an algorithm to find the multiplexer cost. This algorithm runs through all the hardware modules, checks the possible fan-in conditions for all the input terminals of the modules, and calculates the number of tri-state buffers required.

```
for each hardware module, do {
    for each input terminal {
        if (one or zero fan-in) {
            cost no change;
        } else {
            cost += number of total fan-in;
            for each fan-in module {
                if (mux reducible && only one fan-out)
```

```
                cost--;
            }
        }
    }
}
```

Having defined the cost function, we now describe the register file merging algorithm. The register file merging algorithm is basically a simulated annealing approach with some heuristics built in as a preprocess. The purpose of the preprocess is to simplify the problem size so that the annealing time can be reduced. Two heuristics have been considered:

- If two tri-stated registers have the same fan-out and have no I/O conflict, it is advantageous to merge these two registers into a register file.
- If two tri-stated registers have the same fan-in and have no I/O conflict, it is advantageous to merge these two registers.

The second heuristic is not always true. In the cases of data broadcasting, the situation where two registers share the same execution-unit fan-in will also happen; however, these two registers should not be merged. On the other hand, the first heuristic is a useful guidance for the initial clustering and has been built in the algorithm. After implementing the first heuristic, the benchmark results show big improvement on the running time for complicated examples and almost equivalent performance for small examples. The run times for both cases (simulated annealing *with* the initial clustering and *without* the clustering) are listed in Table 2.2. The complete register file merging algorithm is given below.

```
initial clustering of registers based on heuristics;
initializing cost & temperature;
do until satisfied {
    do until satisfied {
        randomly choose a move;
        if no legal move, break;
        evaluate cost;
```

```
        if accept the move,
            update the move matrix and the current cost;
    }
    update temperature;
}
```

To select a legal move, a move matrix is generated. This matrix is a N by N matrix where N is the number of registers under consideration. Each row of the matrix represents a register file and each column represents a register. Consider the matrix entry (i, j), the possible values of this entry include *NodeIn* (meaning that register j is in register file i), *Restricted* (meaning that register j can not be moved to register file i due to I/O conflicts), and *Available* (meaning that moving register j to register file i is a legal move). Notice that only half of the matrix is needed due to the equivalence of the register file structures before and after the row permutations of the matrix. Each column can have one and only one *NodeIn* entry since a register can only reside in one register file. This move matrix provides a simple mechanism to update the register file structure and to randomly choose a legal move.

3. Results of the Proposed Approach

Table 2.2 demonstrates the performance of the algorithm. For simple examples such as Toy, the optimal solution is found by the algorithm. Figure 2.9 shows the optimal register file structure of Toy. In this example, five registers (r0 to r4) and two execution units (e5 and e6) are allocated. Assuming that all the registers are multiplexer reducible and all the execution units are non-reducible, with the given I/O conflicts of the registers, the algorithm finds the optimal solution, which has three register groups. If each register is treated as a separate entry, five I/O data buses for the registers and six tri-state buffers are needed. The optimal structure only needs three data buses and three tri-state buffers.

For the complex examples such as the IIR's and the BPF (Band Pass Filter), improvements on both bus costs and multiplexer costs are achieved. The reason why the multiplexer costs cannot reduce significantly for some examples is that those examples have extensive resource sharing and therefore the multiplexers are unavoidable. Since the multiplexer cost is much more important than the bus cost, the algorithm doesn't improve

Example	Toy	IIR, v1	IIR, v2	BPF	FIR filter	2nd order WDF	Volterra
# of registers	5	34	44	56	18	14	17
# of tri-state buffers reduced	3	3	1	15	1	2	1
# of buses reduced	2	4	11	28	1	1	3
Run time (SPARC 2)	negligible	12 sec	3 sec	39 sec	1 sec	negligible	1 sec
Run time w/o heuristic	negligible	4 min	3 sec	57 sec	2 sec	negligible	negligible

Table 2.2: Benchmark Results for Register File Merging.

the bus cost much. Table 2.2 also shows that the run time of the algorithm on a SPARC station is within seconds for all the examples. Comparing the run time of the algorithm with the preprocess with that of the algorithm without the preprocess, the algorithm with preprocess takes a little longer for the Volterra example. This is due to the fact that the annealing process is so short that the run time of the preprocess becomes more significant than the run time of the annealing. The total run time of this example, however, is very short for both algorithms. For benchmarks such as the BPF and one of the IIR filters, the preprocess plays an important role in reducing the run time. The run time reduces significantly after the preprocess is implemented.

2.6.2 Multiplexer Reduction

1. Motivation and Problem Definition

The goal of multiplexer reduction is to improve area efficiency by removing multiplexers or tri-state buffers. Several mechanisms can be used to remove multiplexers. Merging registers into register files as described in the previous section is one of the mechanisms. Another possibility is to check the commutativity of operators. If operators such as additions have commutable inputs, their input connections can be swapped to reduce the number of multiplexers. Even when the number of multiplexers cannot be reduced, we can still try to find cheaper ways to implement the multiplexers. Figure 2.10 shows how the

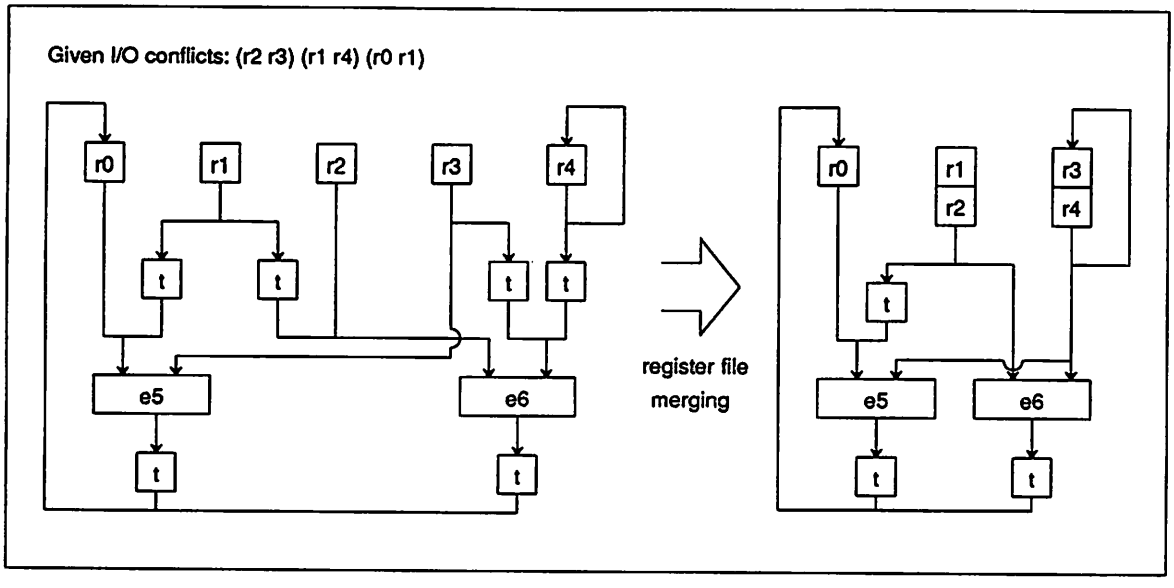

Figure 2.9: Optimal Register File Structure for Toy

combination of these ideas can help to reduce the multiplexer cost. In a continuation of the example shown in Figure 2.3, a hardware graph was generated in the mapping process (shown at left). Assuming that the hardware allocator only allocates one adder, namely adder1, for the algorithm, two multiplexers are required for multiplexing the inputs of the two additions. If Register c and Register d do not have any I/O conflict, they can be merged into a register file and one of the multiplexers can be reduced. Similarly, if Register a and Register b do not have any I/O conflict, they can also be merged and the final hardware structure is shown at the right top corner of the figure. However, if Register a and Register b *do* have I/O conflicts, the multiplexer at the outputs of Register a and Register b cannot be reduced. Assuming that tri-state buffers are much cheaper than multiplexers, the hardware mapper will replace the multiplexer with two tri-state buffers and the final configuration is shown at the right bottom corner of this figure.

2. Approaches

Finding the cheapest way to implement a multiplexer can be a complicated problem. In general, given a cell library with several implementations of tri-state buffers and multiplexers, finding the cheapest way to implement a N-input multiplexer with the given

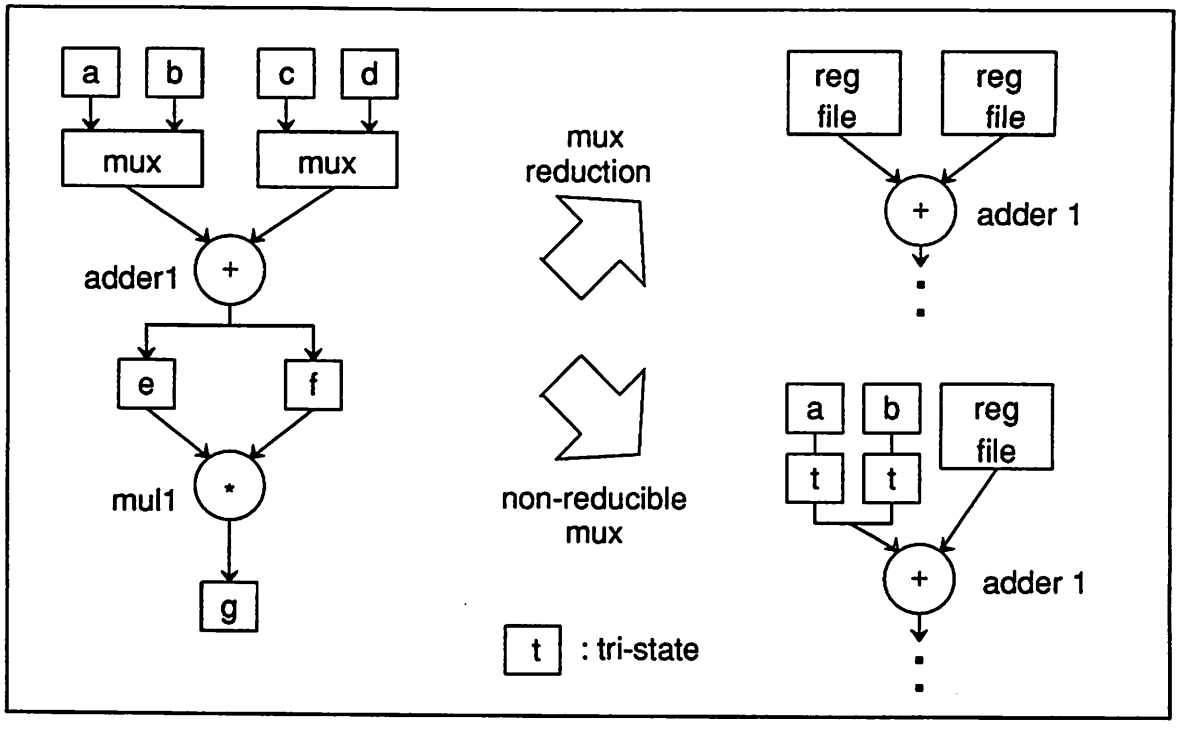

Figure 2.10: Multiplexer Reduction

cells, where N is any natural number, is a combinatorial problem. Fortunately, N is relatively small (less than six) for most cases and an exhaustive search can be performed to find the cheapest implementation. Instead of doing an exhaustive search, a divide-and-conquer search mechanism is developed to find a good multiplexer implementation. This mechanism divides N into two parts, N1 and N2, and go through all possible values of N1 and N2 to find the best cost of the N-input multiplexer. This approach is a recursive algorithm in which the cost of the N-input multiplexer depends on the costs of the multiplexers with fewer inputs. The complexity of this approach is $O(N^2)$. Comparing with the exhaustive search, the simplification is in the division of N into two parts. For exhaustive search, N can be divided into two to N parts and this search leads to a combinatorial problem called *partitions with restriction*[68]. In conclusion, the search for a low-cost multiplexer implementation is important in achieving an efficient design. With the proposed search mechanism, this search is greatly simplified.

Algebraic commutativity can be used to remove multiplexers. Figure 2.11 illustrates the idea. If an operation has commutable inputs, the hardware mapper will perform

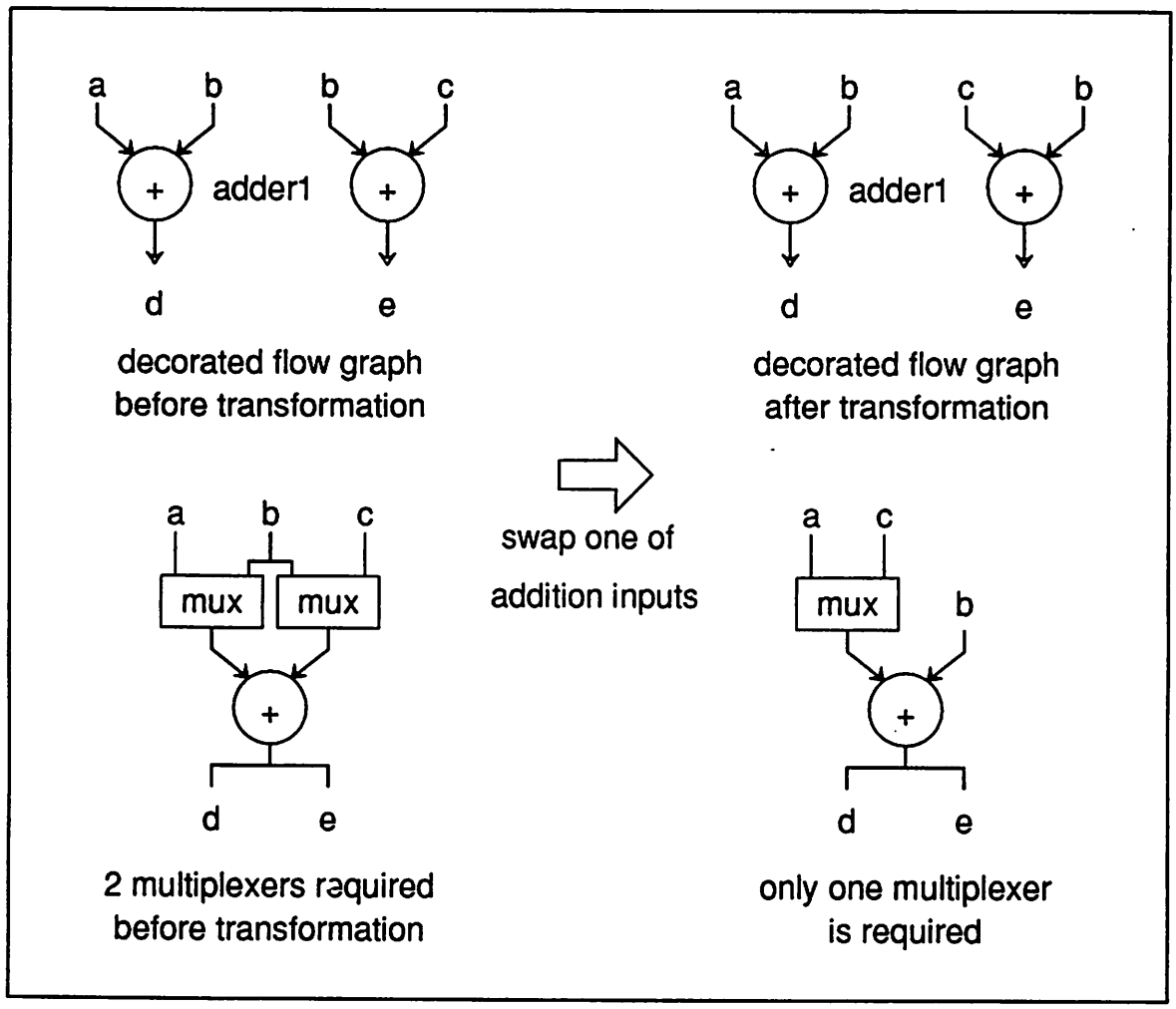

Figure 2.11: Algebraic Commutativity for Multiplexer Reduction

an exhaustive search to swap its commutable input connections and find the minimal number of multiplexers required. The search time is on the order of 2^n where n is the number of fan-in's of the commutable inputs. Under most circumstances, n is less than 10 and the run time is within seconds. Very often, designers won't pay attention to the algebraic commutativity when specifying an algorithm, and since the synthesis process doesn't take the multiplexer cost into account, extra multiplexers may be allocated. The proposed search which is based on the algebraic commutativity provides a simple mechanism to remove these extra multiplexers.

3. Results

Given the costs of tri-state buffers and multiplexers, Figure 2.12 shows the difference in the multiplexer costs between the exhaustive search and the proposed divide-and-conquer approach. The solid lines are the optimal multiplexer costs and the dotted lines are the solutions found by the divide-and-conquer approach. Consider the cost of a N-input multiplexer where N ranges from 1 to 7, Figure 2.12(a) demonstrates that the costs found by the divide-and-conquer approach and the exhaustive search are exactly the same assuming that the cost of a tri-state buffer is 10 and that the cost of a 2-input multiplexer is 15. As a matter of fact, if a cell library has only tri-state buffers and 2-input multiplexers to implement N-input multiplexers, the cheapest N-input multiplexer found by the proposed approach is the optimal implementation.

Figure 2.12(b) shows the cost difference between the proposed approach and the exhaustive search assuming that the cost of a tri-state buffer is 10, the cost of a 2-input multiplexer is 15, and the cost of a 3-input multiplexer is 20. The maximal difference is 10 for both the 5-input multiplexer and the 7-input multiplexer.

Table 2.3 shows several examples in which the algebraic commutativity is used to reduce the number of multiplexers. For the Epsilon processor, very few hardware resources are shared; therefore, the number of tri-state buffers does not reduce after the transformation. For the Viterbi processor and the IIR filter, 36% and 16% reduction is achieved by applying the commutativity transformation.

Table 2.4 shows the total number of tri-state buffers reduced combining all the proposed techniques. For the four examples shown, the reduction ratio ranges from 38% to 12.5%. Since the multiplexer reducibility depends on many factors such as the amount of

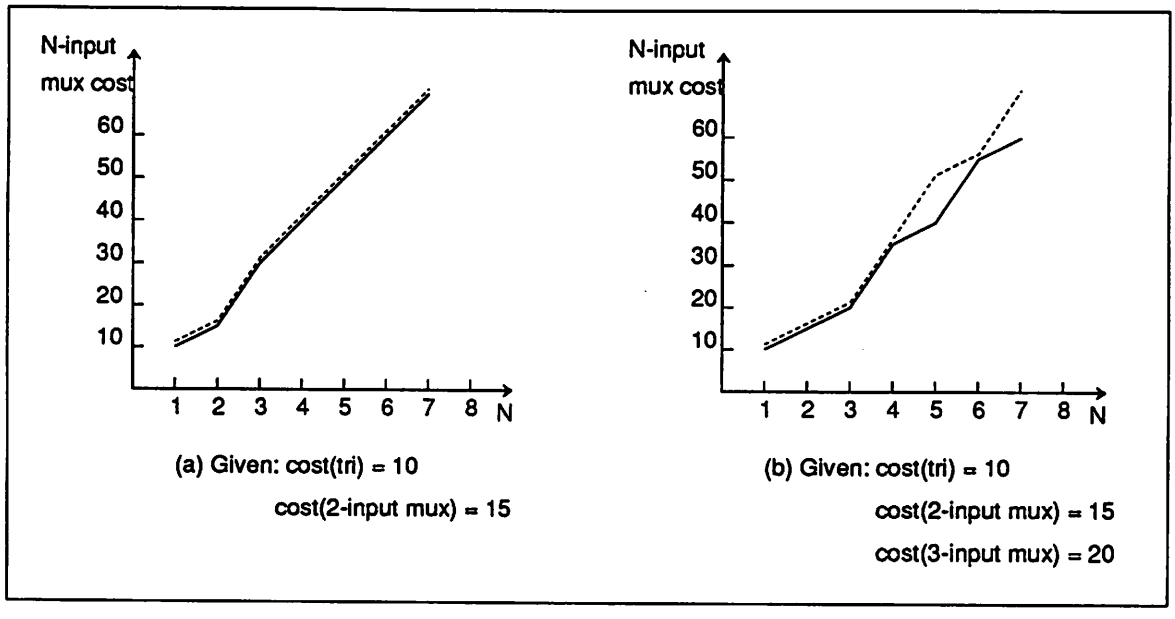

Figure 2.12: Multiplexer Cost Found by Proposed Approach and Exhaustive Search

Examples	Before Transformation	After Transformation
Epsilon	6	6
Viterbi	11	7
IIR	37	31

Table 2.3: Number of Tri-state Buffers Needed Before and After Applying Commutative Transformation

resources shared and the hardware database, the effect of multiplexer reduction varies significantly for different examples. The results in Table 2.4 have demonstrated the importance of the multiplexer reduction for most examples.

2.6.3 Data Path Partitioning

1. Problem Definition

The task of data path partitioning is to divide data-path modules (including registers, multiplexers, and execution units such as adders) into several groups so that a more efficient layout can be achieved. Each group forms a data path and among different groups, data buses are used for interconnect. As described in Chapter 1, HYPER has been de-

Example	IIR, v1	BPF	WDF	Volterra
# of tri-state buffers before reduction	37	39	25	8
# of tri-state buffers reduced	10	15	4	1
Reduction ratio	27%	38%	16%	12.5%

Table 2.4: Total Number of Tri-state Buffers Reduced.

Examples	# of cells	Area improvement	Run time	Note
dp1	3	0%	0.02 sec	find optimal
bdp	13	0%	0.44 sec	
dp2	14	2%	0.48 sec	
odp	23	2%	2.36 sec	
audp	27	-4.8%	5.72 sec	

Table 2.5: Performance of the Linear Placement Program in DPP.

veloped to address the design of the computationally intensive parts of high-performance real-time systems. Therefore, the data path generated by HYPER usually contains a large number of hardware blocks. If all the blocks are put in a single data path, chip layouts can be extremely inefficient. Furthermore, the linear placement program in the Data Path Processor (dpp) of Lager IV, which uses a modified Kernighan-Lin Algorithm [69] [5], can handle up to about 25 hardware blocks well. Beyond the limit, the performance of the program starts to degrade. Table 2.5 shows the performance of the linear placement program in dpp. The area improvement is compared with the placement chosen by human designers. We can see the performance degradation of the program if the data path contains more than 25 blocks. Data path partitioning is therefore necessary to achieve better layouts if there are more than 25 blocks in the data path.

2. Rejectionless Simulated Annealing Approach

The partitioning process is performed in three phases. First, a depth first search is performed throughout the hardware graph. Sets of disjoined blocks are put in different

groups. Each group is further divided if different word lengths are found within the group. This criterion is due to the constraint that Data Path Processor cannot handle multiple word-lengths in a single data path. If the number of blocks in a group is still too large after the above two processes, a modified simulated annealing algorithm [52] is finally used to partition the data paths. The first two processes can be performed in polynomial time [71]; however, the third process which solves the partitioning problem with size constraints is NP-hard [12]. Several partitioning algorithms have been tried for the third phase and the rejectionless simulated annealing approach was finally chosen [52]. The rest of this section will focus on our experience with the partitioning algorithms.

The first approach attempted was the Kernighan-Lin algorithm. The reasons for choosing the Kernighan-Lin algorithm were as follows:

1. For most target applications of HYPER, the total number of data path modules is less than 100 even in an extremely complicated data path.

2. The linear placement program in Lager IV can handle up to 25 blocks pretty well.

3. From 1 and 2, the number of times needed to apply the partitioning process can be derived:

$$100/25 = 4 = 2^2$$

Based on the above analysis, the number of times the two-way partitioning has to be applied for most cases is less than two and therefore a simple partitioning algorithm may be sufficient. However, from the benchmark runs such as various implementations of the IIR filter and the Viterbi processor, we found that the performance (in terms of the number of data buses) of the Kernighan-Lin algorithm is 50% to almost 4 times worse than a simulated annealing approach. Since data buses are extremely expensive in chip layouts, a more sophisticated algorithm with a better performance has to be considered.

Simulated annealing generally performs better than classical edge-interchanging algorithms such as the Kernighan-Lin algorithm [25]. However, the speed and the convergence of simulated annealing can become a drawback when partitioning a large number of nodes with additional constraints such as upper bounds of nodes within a group. A rejectionless simulated annealing approach [52] [58] is therefore adopted and used to deal with both the performance problem of the Kernighan-Lin algorithm and the speed problem

encountered by the simulated annealing approach. This algorithm improves the speed in the following ways:

rejectionless As the traditional simulated annealing process approaches the final solution, most moves generated are rejected and the generation of moves costs most of the run time. The new algorithm improves the run time by accepting all moves. However, the nature of a probablistic approach is not changed since all the moves are ordered according to a badness measure and a move is chosen probablistically.

cooling schedule The traditional simulated annealing approach spends substantial amount of time on evaluating an exponential function to decide if a move is taken. For most computers, exponential functions take much longer run time than additions or multiplications [3]. The new algorithm uses the function $aT_1 + a^3T_2$ for the badness measurement where a is the cost of a move. This function can be easily evaluated and the computer run time can be reduced. At the beginning of the search process, we set $T_1 >> T_2$. Along with the annealing process, we increase T_2 gradually and keep T_1 constant. As time proceeds, the badness measurement gives more preference to moves that are more likely to reduce the cost (more greedy) in an identical way to the simulated annealing approach.

random number generator Random number generators are sources for the simulation of randomness in simulated annealing. It would be interesting to find out how much one can trade on the quality of the random number generators to achieve better speed. Studies showed that a simple linear congruential-based algorithm [53] can improve the speed substantially without sacrificing much quality. The final version of the partitioning program thus uses the Park-Miller random generator for the annealing process.

In the hardware mapper, data path partitioning is performed after the register file merging process. A register file is treated as a macro block and cannot be divided during the partitioning. Some of the hardware modules such as register files and barrel shifters have large areas; on the other hand, some modules are small such as tri-state buffers. Therefore, it is not reasonable to use the total number of modules within a group as the constraint for partitioning. A more reasonable measure is the maximal length of a data path. We

[3] On a SPARC station, an exponential function takes about 5 times longer than a multiplication.

restrict the length of a data path so that extremely long data paths are avoided. Another experience which was gained from inspecting chip layouts is that non-uniform lengths of data paths tend to produce inefficient placement. Therefore, we also try to equalize the lengths of data paths in performing the partitioning.

The annealing process can be divided into two phases: the clustering phase and the exchanging phase. Initially, each node forms a group and the number of groups is equal to the number of nodes. During the clustering phase, nodes are clustered into bigger groups and the number of groups reduces. After the clustering phase, the annealing process enters the second phase (the exchanging phase) when the number of groups remains unchanged and nodes are tossed among different groups.

The constraints for both phases are the same (the maximal length of the data paths) except that some *cushions* are left in the clustering phase for further improvement. This cushion will be reduced with the annealing process and set to zero at the end of the process. The reason for setting the maximal-length constraint with some cushion is that as the clustering process proceeds, the groups with more blocks will tend to attract even more blocks due to their larger *gravity*. Without the constraint or the cushion, the result of the clustering will be several overfed groups with no room to improve. Therefore, we have to be careful not to overfeed these groups in the clustering process.

The rejectionless simulated annealing approach is implemented as follows: a node is first randomly chosen, then a non-empty group is selected based on the badness measure of the node in that group. A group with a smaller badness measure is more likely to be chosen. When a tie occurs (i.e. equal badness measure for several groups), the node being tossed will go to the group with a shorter length to equalize the group sizes. The annealing process stops when further tossing does not improve the cost for a number of moves or when a maximal number of steps (which is a function of the graph size) is reached. A simple outline of the partitioning program is given below:

```
Decide maximal number of steps;
Form a list of bad elements for each groups;
while (stopping criteria not satisfied) {
      Pick a random element a;
      Pick a new group membership for a;
```

Example	Number of nodes	Number of nets	Cost of RSA	Cost of KL
Viterbi	40	51	4	6
IIR, v1	78	109	5	19
IIR, v2	63	93	4	15

Table 2.6: Performance of Rejectionless Simulated Annealing vs. Kernighan-Lin.

```
        (all picks according to probabilities and badness)
        if the move is legal (not violating maximal-length constraint)
           Update graph, cost, and bad lists;
}
```

3. Results

Table 2.6 compares the rejectionless simulated annealing (RSA) approach with the Kernighan-Lin (KL) algorithm. The performance of the RSA approach is much better than the Kernighan-Lin approach for the three examples shown. Since the Kernighan-Lin approach is implemented in Lisp and the RSA approach is implemented in C, it is unfair to compare the run time. However, the run time is not a major issue in the comparison since the run time for both approaches is less than 20 seconds. Table 2.7 compares the performance of the rejectionless simulated annealing approach with the traditional simulated annealing (SA) approach. For the simple examples such as EX1 and EX2, both approaches have similar performance. For complex examples, the new approach either has much shorter run time (FIR and IIR, v3) or much better performance (Volterra and IIR, v1). From the comparison, we are confident about the performance and the run time of the rejectionless simulated annealing approach. Notice that for the third IIR example, the minimal costs found by both approaches are still very high. This is due to the tightly-connected structure of the graph.

Example	# of nodes	# of nets	Cost of RSA	Steps of RSA	Run time of RSA	Cost of SA	Steps of SA	Run time of SA
EX1	10	12	7	4101	1sec	6	5310	1sec
EX2	10	12	4	3065	0sec	4	5310	2sec
IIR, v1	78	109	5	32147	15sec	11	55770	4sec
IIR, v2	63	93	4	25491	12sec	5	46748	3sec
IIR, v3	67	116	31	20420	8sec	30	1998777	117sec
FIR	40	61	9	35002	3sec	13	3000223	177sec
WDF	39	56	9	35002	3sec	14	41170	4sec
Volterra	65	97	3	35003	9sec	9	54017	3sec

Table 2.7: Performance of Rejectionless Simulated Annealing vs. Traditional Simulated Annealing.

2.7 Control Path Generation

After the data paths are generated, the hardware mapper proceeds to generate the control paths, which include a finite-state machine (FSM) as the central control and several control slices as the local control and the interface between the FSM and the data paths. Figure 2.13 illustrates the major steps in the control path generation. Before the FSM and interface logic can be generated, a state transition digram (STG) is extracted from the scheduling information of the decorated flow graph. The state transition diagram is optimized by removing some dummy states in which no operations are performed. Examples of the dummy states are the ending states of the *if* control macros. The optimized STG is then used in both the FSM map generation and the register file merging process (described in the data path generation). An FSM map is a 2-dimensional table in which each control signal is assigned a Boolean value[4] for each state. The FSM map is constructed from both the structure of the interface logic and the state transition diagram. This map will be further optimized by several control transformations to reduce the area of the control paths and the size of the control nets. Finally, a bds description [72] for the FSM and several bds and sdl files for the interface logic are generated.

In general, control signals can be categorized as control inputs and control outputs. Control inputs are signals generated by the FSM to control the data paths or the memory modules. Control outputs are signals generated by the data paths and used by the FSM to

[4] A Boolean value can be 0, 1, or DONT_CARE.

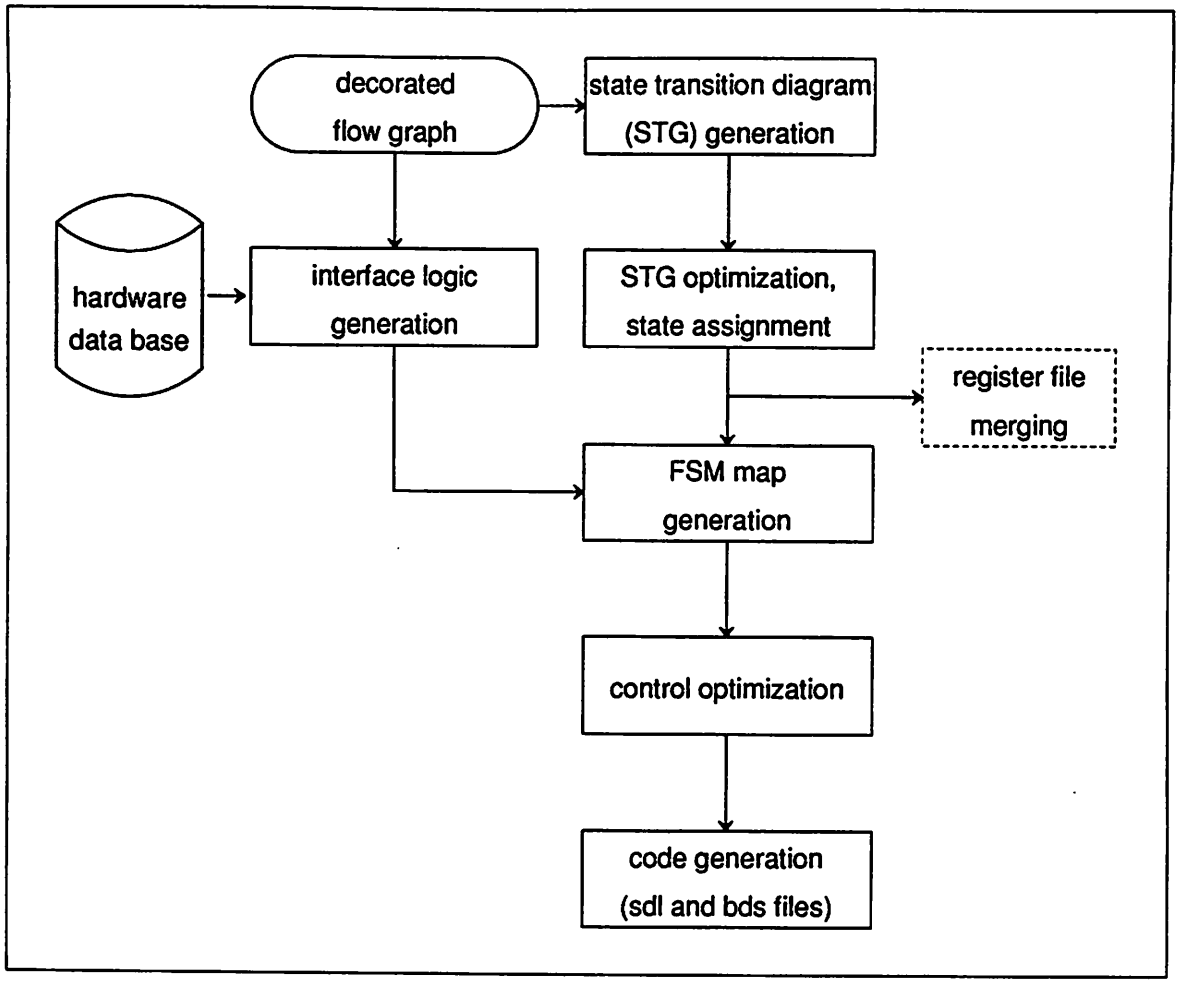

Figure 2.13: Major Steps of Control Path Generation

make decisions on state transitions. For the control inputs, the required interface logic can be generated by consulting the hardware database. The CTL-IN-TERMINAL attribute in the database specifies the values of the control inputs given the operation to be performed. From these values, the required interface logic can be derived. For the generation of the control outputs, not only the hardware database needs to be consulted, the decorated flow graph must also be traced. By simply consulting the hardware database without tracing the flow graph, redundant logic may be generated. For example, the overflow logic of an adder may not be needed for certain instances of the adder. Therefore, the interface logic generator uses a *demand-driven approach* by tracing the flow graph to avoid redundant logic.

The state diagram generation is described first in this section, followed by the

FSM and interface logic synthesis. Finally the control optimization is discussed.

2.7.1 State Transition Diagram Generation

A State Transition Diagram (or State Transition Graph STG) is a graph in which each node represents a state and each edge represents the transition between its input node (called the current state) to its output node (called the next state). Each edge is associated with a condition which specifies when the transition should take place. Each node (i.e. state) is associated with some operations to be performed in that state. A node can also be an idle state in which no operation is performed.

State transition diagrams can be extracted from the scheduling information of the decorated flow graphs. The schedule of a node is expressed by a non-negative integer interpreted as the relative ordering of the node to the other nodes in the same subgraph. Figure 2.14 shows a simple example to illustrate the STG generation process. Given a decorated flow graph at the left, this process generates a STG as shown at the right. The state with a double-circle is the initial state. Since there is no control macro in the flow graph, all the transitions in the STG are unconditional. Notice the transition from State 3 back to State 0. This is due to the Silage assumption that all Silage descriptions are recursive. That is, there is always an infinite loop outside of the whole Silage description.

State diagram generation is a recursive process due to the hierarchical nature of the flow graph. To process a flow graph, the STG generator first groups nodes into states based on the scheduling information and then generates states according to several generation rules. Figure 2.15 illustrates these rules. A STG structure as shown at the left hand side of the figure will be generated if a *loop* node is found in the flow graph. State s represents the state in which the looping condition is checked. During the first iteration, the FSM will transfer to the states corresponding to the subgraph of the *loop* node without going through the State s. Each time after the subgraph is performed, the looping condition will be checked to determine whether the FSM will stay in the loop. If this condition is true, the FSM stays in the loop; otherwise, the FSM jumps out of the loop and goes on to the state corresponding to the next macro node of the flow graph.

The STG structure of the *if* nodes is shown at the right hand side of Figure 2.15. In State s, the branching condition of the *if* node is checked to determine which branch to take for the FSM. Each branch of the STG is represented by a subgraph of the *if* node. A

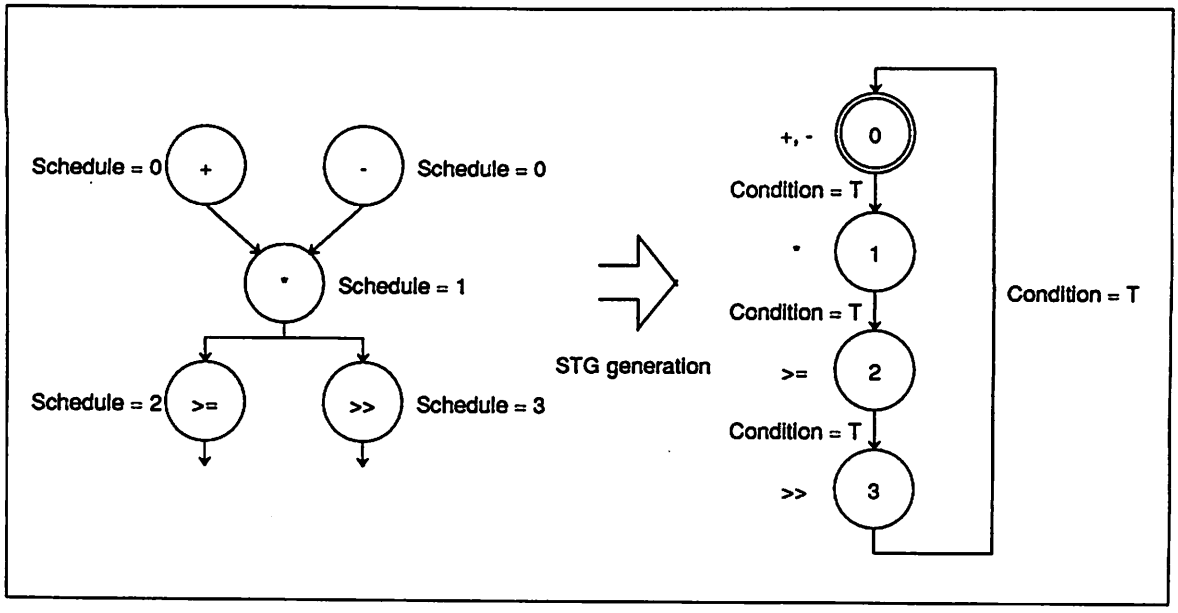

Figure 2.14: State Transition Diagram Generation

dummy ending state is generated in the STG as the merging state of the branches. This dummy state will be removed later in the state optimization process.

Figure 2.16 shows a simple example of the STG generation process using the above generation rules. For clarity, each state in the STG is properly labeled with the function of its corresponding node in the flow graph.

After the state diagram is generated, a transformation is performed to remove the dummy states of the flow graph. Dummy states are states in which all hardware blocks, including execution units and memories, are idle. Dummy states do not include states for synchronization such as *wait* states. An example of dummy states is the ending state of the *if* control macro. Due to cases such as an *if* macro inside another *if* macro, the transformation process has to trace the state transition diagram to find out the correct next states for all the states after the transformation. Figure 2.17 shows a simple example of the state transition diagrams before and after the transformation. In this figure, the *fi* states represent the ending states of the *if* macros. Notice that the *s* state becomes the real ending state of both *if* macros after the transformation.

After the transformation, a straightforward state-assignment strategy is applied to the state transition diagram. The number of bits required for the states using this strategy is $\lceil log_2(number\_of\_states) \rceil$. The state-assignment process can also be performed by

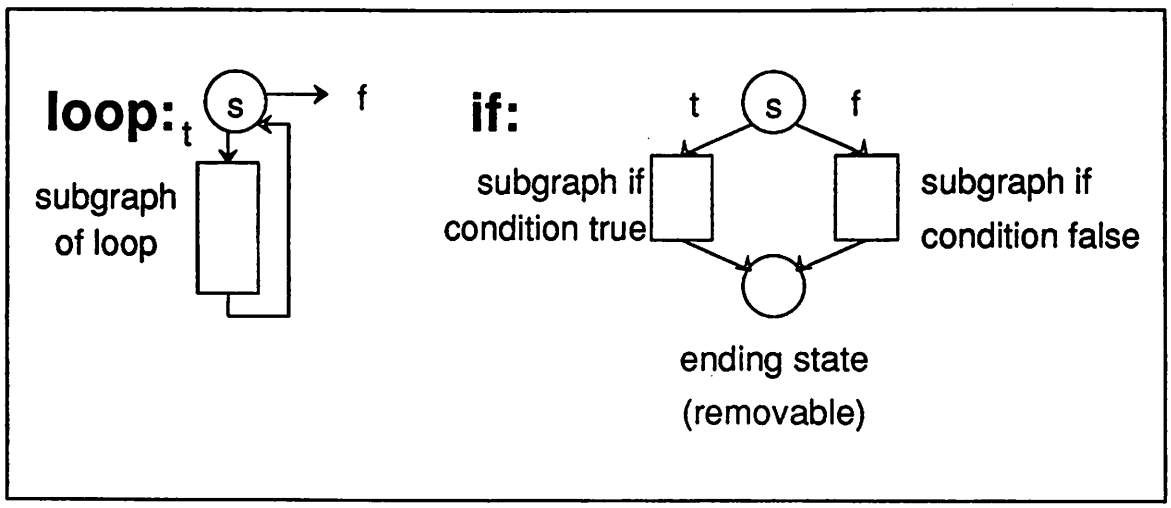

Figure 2.15: Rules for State Transition Diagram Generation

Examples	Number of Inputs	Number of Outputs	Number of States	Area Reduction
Epsilon	9	18	10	7.69%
Viterbi	16	31	20	15.6%
7th order IIR	4	74	12	0%

Table 2.8: Area Reduction Achieved by NOVA

programs such as NOVA [4] or JEDI [3]. NOVA is a program that performs an optimal assignment of binary codes to the states of an FSM. JEDI is a general symbolic encoding program intended for multi-level logic optimization. Since the implementation of the central controller in HYPER is a PLA based FSM, NOVA is chosen as the potential candidate for state assignment to improve the FSM area. Table 2.8 shows the area reduction after the state assignment performed by NOVA. For more control-oriented examples such as the Viterbi processor, a 15.6% area reduction is achieved. However, for many other target applications of HYPER such as the IIR filter, the state transition is purely sequential and no area reduction is achieved by NOVA. In conclusion, more elaborate state-assignment program should be involved in the hardware mapper to replace the straightforward state-assignment strategy currently implemented when synthesizing more control-oriented applications.

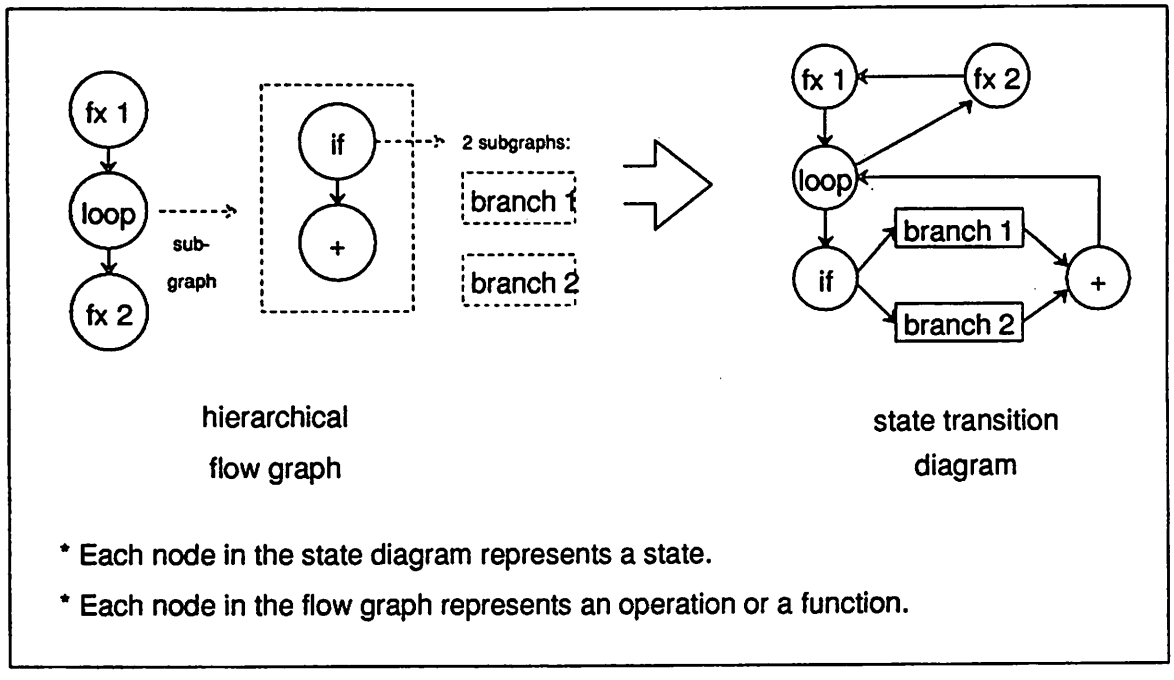

Figure 2.16: An Example of State Transition Diagram Generation

2.7.2 Control Slice Synthesis

Control slices (or interface logic) between the data paths and the FSM are purely combinational with no state. Therefore, the interface logic can be directly generated from the decorated flow graph without the scheduling information. As described before, the interface logic contains two parts: control logic required by signals from the FSM or outside the processor to the data paths (called control inputs) and control logic required by signals from the data paths to the FSM or off chip (called control outputs). For control inputs, the hardware database is consulted to find out what logic is required for each hardware block. For control outputs, the required logic is produced by a demand-driven algorithm so that no redundant logic is generated.

The interface logic is partitioned according to the partition of the data paths. The purpose of the interface logic partitioning is to improve the layout efficiency. We have experimented to put all the interface logic in one module and this configuration produced a routing bottleneck around the interface logic. With the interface logic properly partitioned, the floor planning and the routing of the whole processor can be performed much more efficiently.

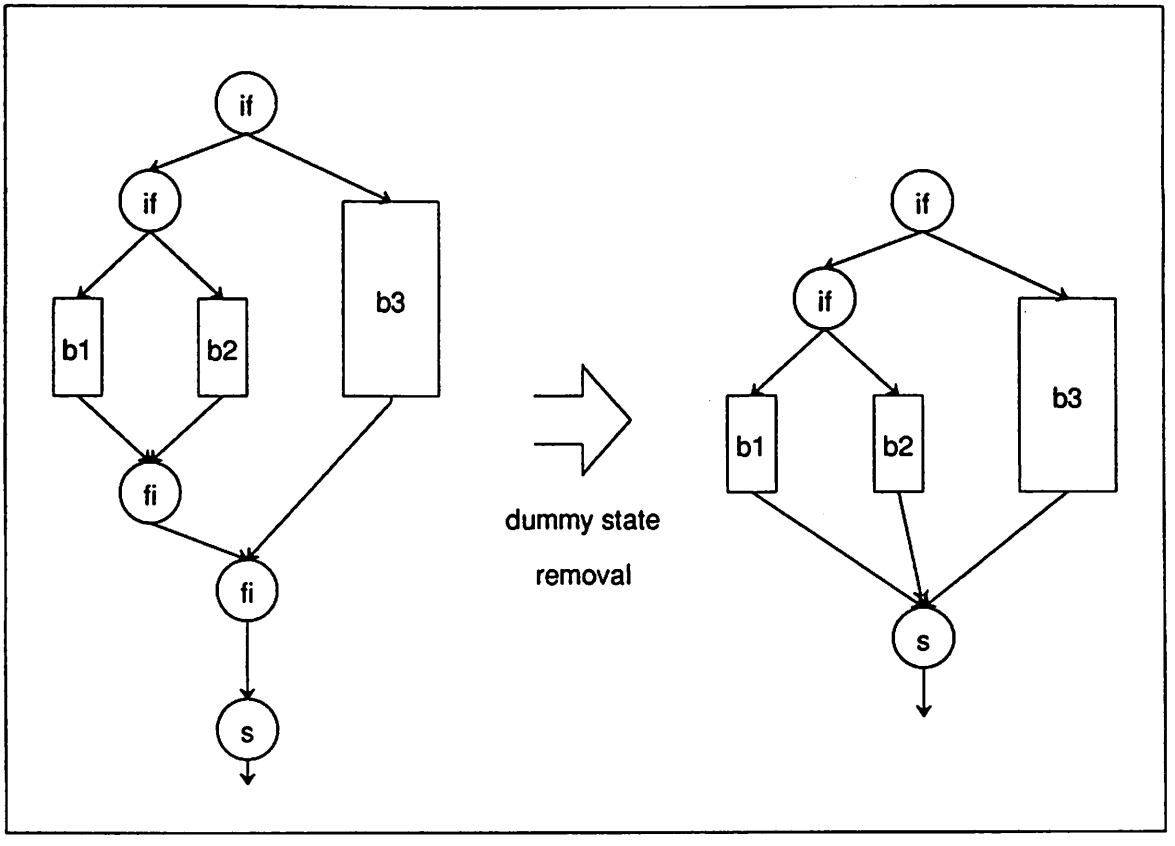

Figure 2.17: Dummy State Removal

The best strategy for the layout of the interface logic would be pitch matched with the data path blocks. The resulting routing between the data paths and the interface logic becomes very clean without any *doglegs*. Figure 2.18 shows three types of control-slice structures. In this figure, two data paths and their interface logic are shown. In (a), the interface logic is not partitioned. In (b), the interface logic is partitioned but not pitch matched with the data-path blocks. In (c), the interface logic is partitioned and pitch matched with the data-path blocks. (c) produces the best layouts among the three and is the intended structure. However, the Lager IV environment currently cannot support the pitch matching between the data paths and the control slices. Therefore, all the layout examples that have been generated from HYPER have the structures as shown in (b).

A bds file and a sdl file are generated for each interface logic module after the control optimization step, which are described later. MIS II further reduces the logic in the layout generation process. A standard cell implementation is finally generated through

Lager IV with each control terminal properly assigned to an edge of the interface logic module.

2.7.3 Finite-State Machine Synthesis

During the control slice generation process, an FSM I/O list, which specifies the input and output signals of the FSM, is produced. The I/O list along with the state transition diagram is used to generate a FSM table (or FSM map), in which the value of each control signal at each state is specified. The entry of the FSM table can be one of the following three Boolean values: TRUE, FALSE, and DONT_CARE. Each entry is assigned a default value initially. The correct values of the entries will be determined later by going through the state transition diagram and activating operations in each state. One heuristic for the FSM synthesis process is to assign the Boolean value DONT_CARE to the FSM table entries as much as possible. This facilitates the subsequent control optimizations as well as increases the degrees of freedom for the logic synthesis tool MIS II. The control signals for registers (LOAD and OUTPUT_ENABLE), however, should never be DONT_CARE to guarantee the correctness of the data flow.

Since registers are allocated to edges (not nodes) in the flow graph, the register LOAD/OUTPUT_ENABLE operations are not part of the operation lists associated with the states in the state transition diagram. The operations of a register are activated by its input/output execution units and by tracing the flow graph. The LOAD/OUTPUT_ENABLE signal is activated only when its input/output execution unit is activated.

As described in the previous section, multiplexers/tri-state buffers will be properly introduced during the data path generation process. The values of the selection signals of these multiplexers/tri-state buffers are determined by the trace of the *hardware* graph. The hardware graph is traced in the direction of the data flow in the flow graph. Figure 2.19 illustrates the idea. When Operation A is activated, data flows from the input of A (i.e. reg1) to the output of A (i.e. reg2). The corresponding hardware graph is traced from reg1 to reg2. Three control signals are turned on during the trace: the OUTPUT_ENABLE (oen) of reg1, the select signal of mux1, and the LOAD signal of reg2.

After all the entries in the FSM table are specified, we can start performing the control optimizations to reduce the table size and to simplify the routing between the control path and the data paths.

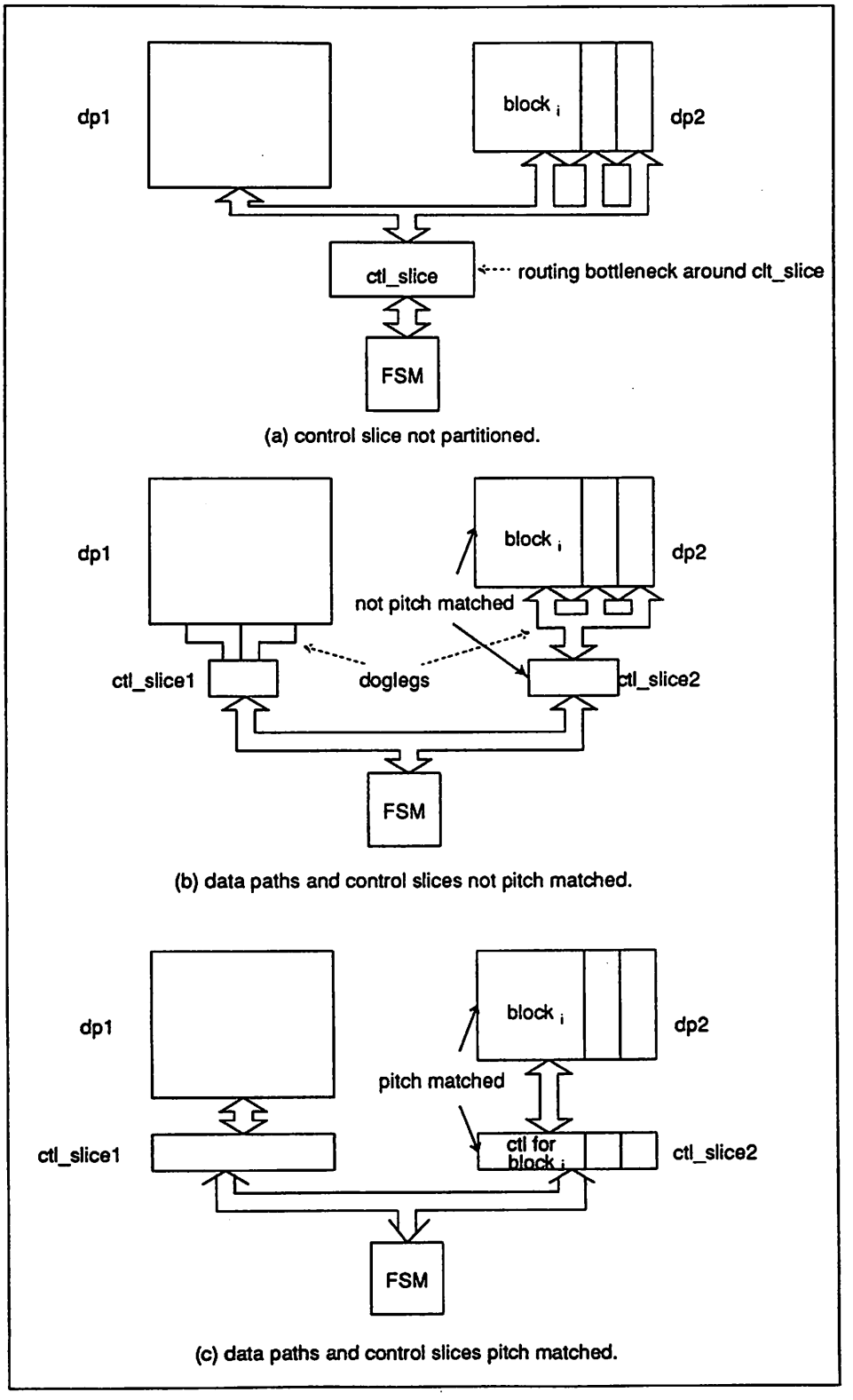

Figure 2.18: Three Types of Control Slice Structures

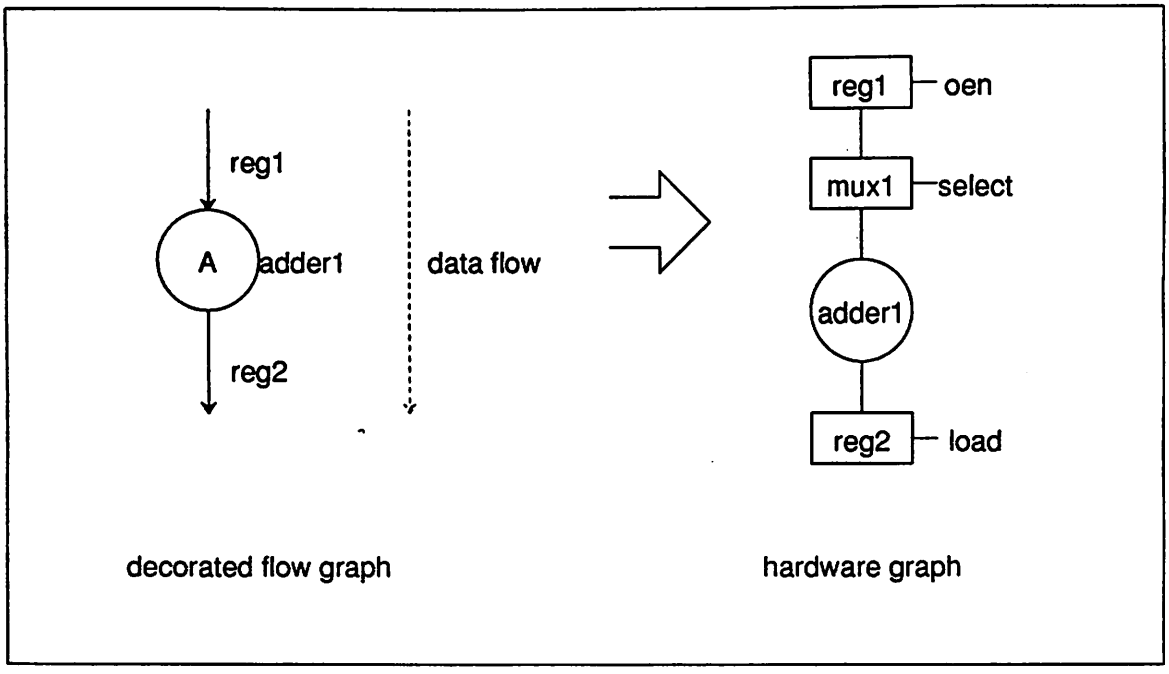

Figure 2.19: Trace of the Hardware Graph

2.7.4 Control Optimization

The control path synthesis process in HYPER performs four transformations on the interface logic and the FSM to reduce the control path and the control net routing. These four transformations are described below with some simple benchmark results showing the significance of the transformations. The overall performance of these optimizations is discussed at the end of this section.

1. Merging Equivalent or Complementary Signals

The first control optimization in the control path synthesis process is to merge equivalent or complementary FSM control signals. This optimization may require proper allocations of buffers or inverters in the interface logic. As described above, the Boolean value *DONT_CARE* is assigned as much as possible in the FSM synthesis step to facilitate the merging. For example, if an execution unit is not activated at a particular state, all its control inputs are set to DONT_CARE. With this optimization, the FSM size and the control-net routing are reduced substantially due to the DONT_CARE assignment and the fact that many control signals are inherently complementary to each other. Table 2.9 lists the FSM sizes before and after the optimization from some benchmark runs. The complexity

	Epsilon processor	Viterbi processor	7th order IIR	11th order FIR	5th order WDF
# of flow graph nodes	24	51	44	31	116
# of flow graph edges	40	100	61	41	144
# FSM outputs before merge	22	48	93	60	184
# FSM outputs after merge	17	28	76	35	161
reduction percentage	22.7	41.7	18.3	41.7	12.5

Table 2.9: Benchmark Results for Control Signal Merging

of each benchmark is also shown in this table. The same benchmarks are used throughout this section and the complexity will not be repeated in the other tables. The result shows 12.5% to 41.7% FSM size reduction from this optimization and the run time is negligible for all benchmarks.

Notice that MIS II also performs the merging of equivalent or complementary signals. The reasons for performing this transformation in the control optimization step rather than performing in MIS II are: first, not as many formal terminals[5] are produced for the FSM. Second, the hardware mapper can properly allocate buffers in the interface logic.

2. Local Control or No Control

The second control optimization in the HYPER control path synthesis is to recognize control signals that are independent of the FSM states and replace them by a distributed control in the interface logic. This optimization is especially useful for cases such as pipeline registers and multiplexers since the load and output-enable signals of pipeline registers can be simply driven by clock signals and the control signals of multiplexers can usually be hard wired locally. Table 2.10 shows the FSM reduction result from this optimization. 8.6% to 25% reduction is achieved.

[5]In OCT, a *formal terminal* is a terminal of the current facet and an *actual terminal* is a terminal of an instance. That is, a formal terminal is an I/O terminal in the current block, whereas an actual terminal is a terminal internal to the block.

	Epsilon processor	Viterbi processor	7th order IIR	11th order FIR	5th order WDF
# FSM outputs before opt.	22	48	93	60	184
# FSM outputs after opt.	19	42	85	45	166
reduction percentage	13.7	12.5	8.6	25	9.8

Table 2.10: Benchmark Results for Utilizing Local/No Control Optimization

	Epsilon processor	Viterbi processor	7th order IIR	11th order FIR	5th order WDF
# FSM outputs before opt.	22	48	93	60	184
# FSM outputs after opt.	22	44	85	50	140
reduction percentage	0	8.3	8.6	16.7	23.9

Table 2.11: Benchmark Results for Optimizing Register-File Control Signals

3. Register File Decoder Allocation

Allocating decoders for register files can reduce FSM sizes as well as the wiring between FSM's and interface logic. Table 2.11 shows the benchmark result for this optimization. The performance of this optimization depends on the quality of the register file merging in two ways. The reduction is more significant if more register files are generated and this optimization is more important when the register file sizes are large. Table 2.11 shows that the reduction has no effect on one of the benchmarks, the Epsilon processor. This is due to the fact that the Epsilon processor is a relatively simple example and therefore it has very few registers allocated. Since no register file is generated for the Epsilon processor, the reduction rate is 0%. On the other hand, this optimization achieves significant reduction for some other examples such as the wave digital filter. This wave digital filter has 53 registers allocated and all the registers have the same word length. Therefore, the register file merging can be performed efficiently.

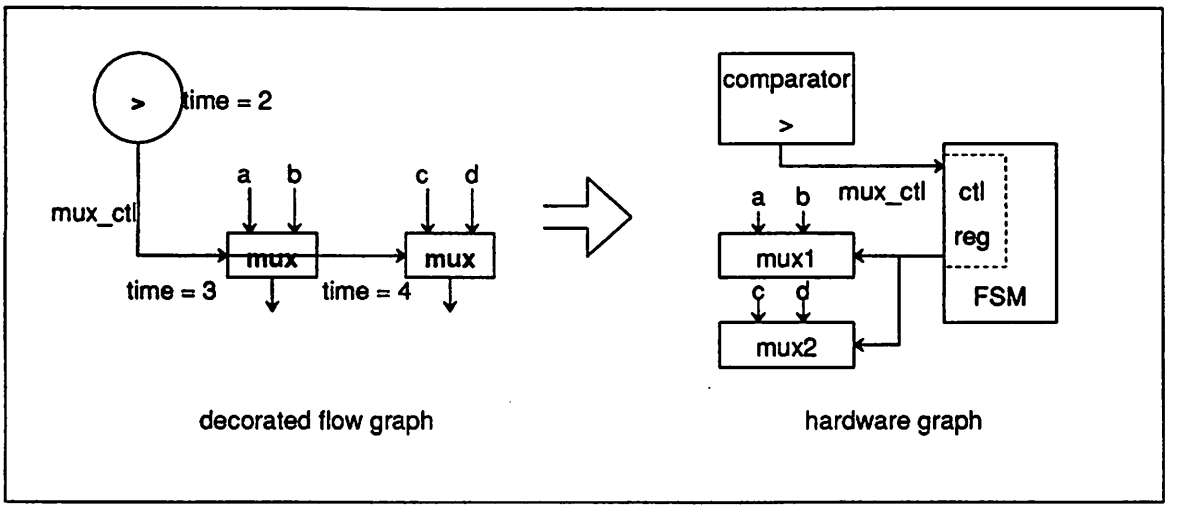

Figure 2.20: Control Register Allocation

4. Control Register Allocation

The final optimization in the control path synthesis uses the life time analysis and the algorithm for the clique partitioning problem [13] to allocate the minimal number of control registers. Control registers are needed when the producing time and the consuming time of a control signal are not equal, and therefore a temporary storage is required. Control registers are not allocated in the HYPER hardware allocation process since the hardware allocator only allocates data-path registers. Control registers will be part of the finite-state machine and are treated identically to the states of the FSM. Figure 2.20 shows a simple example to illustrate the allocation of control registers. In this Figure, Mux_ctl is generated at Time 2 and consumed at Time 4. Between Time 2 and Time 4, a control register in the FSM is allocated to store the value of mux_ctl.

Figure 2.21 demonstrates the procedure of allocating the control registers. In Step 1, the life time of all control signals is collected. In Step 2, a conflict graph G is derived from the life time collected in Step 1. In G, each node represents a control signal and an edge is drawn between Node i and Node j if the life time of control signal i and control signal j overlaps. In Step 3, the complementary graph G' of G is derived. In G', an edge is drawn between Node i and Node j if there is no edge between Node i and Node j in G. Finally in Step 4, the algorithm for the clique partitioning problem is applied to G' and in this example, two cliques are found. We conclude that two control registers are needed to

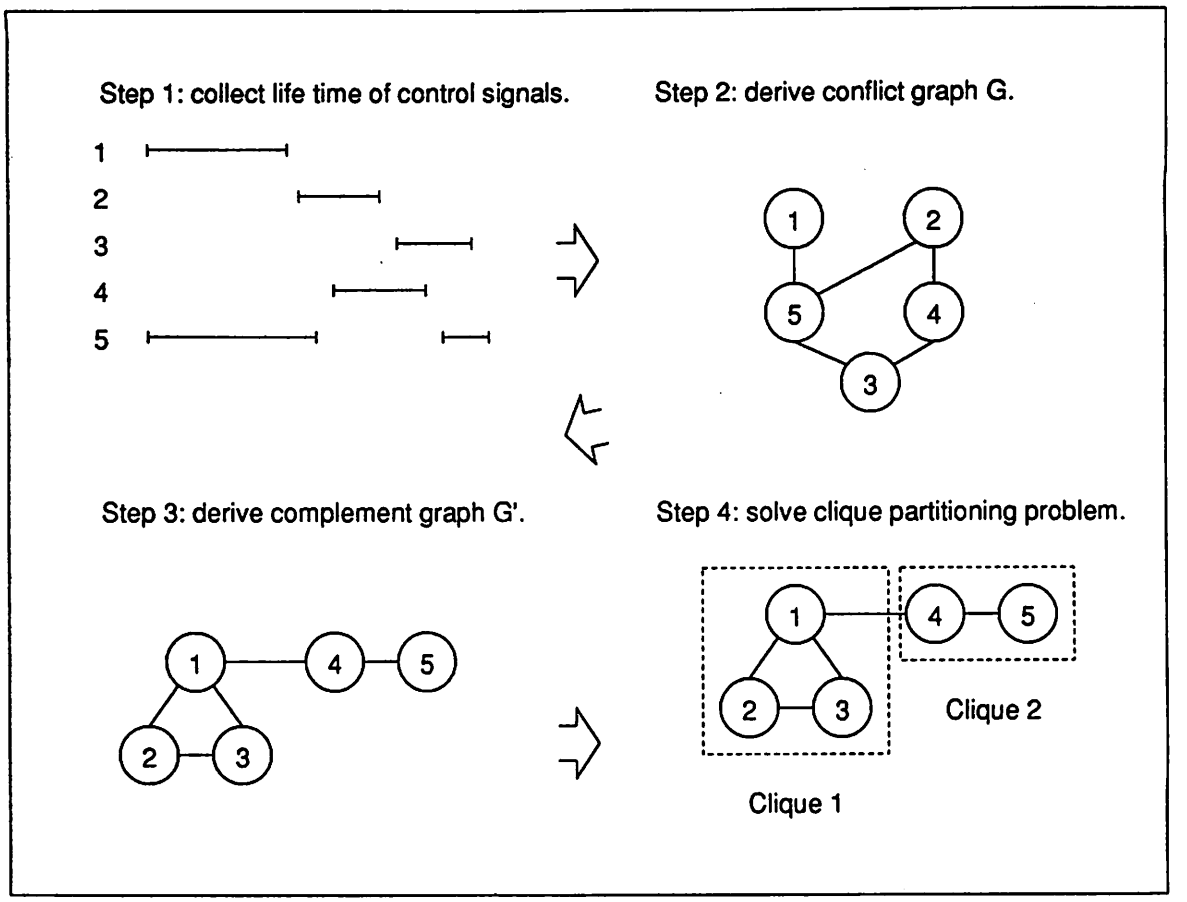

Figure 2.21: Procedure for Allocating Control Register

store the five control signals in this example.

This optimization does not improve FSM sizes much. For the IIR, FIR, and WDF examples, there is neither control signal nor control macro description in the flow graphs. Therefore, no control register is needed. For the Epsilon processor and the Viterbi processor, very few control registers are required and the optimization doesn't reduce the FSM much either. The exact numbers of the control registers required and reduced depend on the scheduling of the control operations that generate and/or consume the control signals. This optimization results in approximately one to two bit reduction of the FSM states for both the Epsilon processor and the Viterbi processor. Similar results are obtained for the CORDIC algorithm since the CORDIC example doesn't have many control operations either. When dealing with examples that involve many control macros and many control-signal operations

	Epsilon processor	Viterbi processor	7th order IIR	11th order FIR	5th order WDF
# FSM outputs before opt.	22	48	93	60	184
# FSM outputs after opt.	14	26	66	32	134
reduction percentage	36.4	45.8	29.0	46.7	27.2

Table 2.12: Benchmark Results for Overall Control Optimizations

such as comparisons, this optimization can become very useful.

5. Overall Performance of Control Optimization

The overall performance of the control optimizations is shown in Table 2.12. The optimization process reduces FSM sizes substantially. 46.7% to 27.2% reduction is achieved for the benchmarks.

2.8 Significance of the Transformation Order

The major transformations performed in the data path synthesis process are register file merging, multiplexer reduction, and data path partitioning. The natural order to perform these transformations is to merge registers first, taking into account of the multiplexer cost, then perform the multiplexer reduction. When the hardware graph is finalized with register files treated as macro nodes, the partitioning process partitions the hardware graph and generates the data paths.

These transformations are all ad hoc transformations. They are performed as the synthesis process proceeds. Therefore, the order of the transformations is fixed according to the synthesis steps. Without these transformations, the synthesis result is still correct. However, the area efficiency will be much worse.

When performing several transformations in sequence, these transformations should be orthogonal. Otherwise, the transformations performed first should take into account of the effect of the subsequent transformations. In the hardware mapper, data path partitioning is orthogonal to register file merging and multiplexer reduction. However, the register

file merging is not orthogonal to the multiplexer reduction. Therefore, when merging registers into register files, multiplexer costs are calculated based on the expected multiplexer structure *after* the multiplexer reduction.

Unlike the data path transformations, there is no inherent order for the control path transformations. Currently, under the HYPER hardware mapper, the order of these transformations is to perform merging equivalent/complementary signals first, followed by utilizing local control, then allocating register file decoders, and finally the control register allocation. The first three transformations are mainly for reducing the FSM output size, while the last transformation deals with the number of FSM states. These two categories of transformations are independent from each other and can be performed in either order without affecting the synthesis result[6]. Within the first category (i.e. the first three transformations), the order of the transformations doesn't affect the final FSM size either. The reason is that all the transformations check on certain properties of a control signal. If the control signal has such a property, the transformation tries to reduce this signal. For signals with one or more of the properties, they will be reduced anyway regardless of the order of the transformations. For signals without any of the properties, they will still be part of the FSM after the transformations, also regardless of the order of the transformations. Table 2.13 shows the FSM sizes of the same benchmarks as the previous section with a different control optimization order. This result confirms our reasoning that the order of the control optimizations doesn't affect the final FSM sizes.

Even though FSM sizes will not be affected by the order of the transformations, the final structure of control paths may be slightly different for various orders of the control transformations. For example, if we perform the transformation of utilizing the local control *before* the transformation of merging equivalent control signals, a signal with both properties (i.e. the FSM entries of this signal are both independent of the states and equivalent to the entries of another signal) will be produced differently. This signal will be generated from a local control slice rather than from the fan-outs of another control slice. Figure 2.22 shows the difference of the interface logic for these two different transformation orders. The tradeoff between these two structures is that the first structure has simpler interface logic,

[6]If a more sophisticated state-assignment program such as NOVA is performed, the order of the control transformations still doesn't affect the final result. The reason is that these transformations try to reduce either the number of states or the number of control outputs. However, the state assignment tries to reduce the PLA (or FSM) size given the number of states and the number of control I/O's. These control transformations should be performed before the state assignment so that the minimal number of states and control outputs are given to the state assignment process.

	Epsilon processor	Viterbi processor	7th order IIR	11th order FIR	5th order WDF
# of FSM outputs before opt.	22	48	93	60	184
# of FSM outputs after opt. order 1– > 2– > 3	14	26	66	32	134
# of FSM outputs after opt. order 2– > 1– > 3	14	26	66	32	134

Table 2.13: Benchmark Results Showing the Effect of Different Control Optimization Orders

while the second structure has simpler routing for the control nets. Techniques have been proposed in [44] and [55] to transform Structure 1 to Structure 2 so that little replication is traded for a smaller number of partitioned nets.

2.9 Processor Synthesis

After data paths and control paths are generated, the top level processor can be produced. The major tasks in the processor synthesis are memory block generation, net generation, and processor optimizations which try to improve layout efficiency and processor performance.

To generate memory blocks, the synthesis program first consults the array database to find the hardware parameters for each block. If a parameter value is not specified by the user or by the previous synthesis steps, the hardware mapper will try to derive the value from other relevant parameters. If the parameter value still cannot be determined after the derivation, the user has to specify it in the layout generation phase.

The net generation process produces all the interconnects between modules including data, control, clock, and power nets. To ensure the electrical performance of the processor, local buffers are allocated for the clock nets to avoid possible clock skew. Furthermore, the sizes of these buffers are properly scaled according to the *stage ratio* [51] to achieve the minimum delay[7].

[7]For a cascaded set of inverters, the optimum stage ratio for minimum delay is 2.7. However, ratios of 2 to 10 can be used to optimize other attributes such as size or power dissipation.

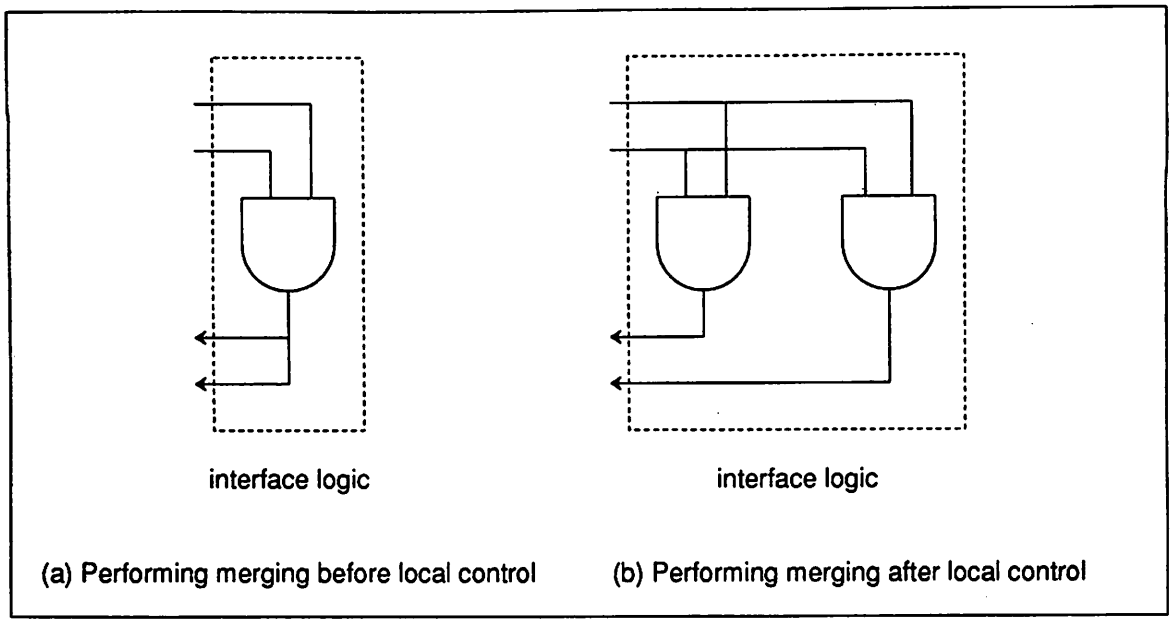

Figure 2.22: Different Interface Logic Structures As Resulting From Two Control Optimization Orders

Based on the study-and-critique of the final layouts of some benchmarks such as the 7th order IIR filter and the Viterbi processor, the hardware mapping process has been gradually refined. Especially the heuristics of the data and control path partioning have been heavily influenced by this generate-critique process. This process has resulted in a dramatic improvement in the area efficiency of the layouts generated by the hardware mapper. The most important heuristics are listed as follows:

- Interface logic should be properly partitioned. The standard cell implementation should be specified as one row and, if possible, pitch matched with the data path blocks.
- Data paths should be partitioned into approximately equal sizes. Data paths of irregular sizes usually produce inefficient layouts at the processor level.
- Both data terminals and control terminals of data paths and control paths should be properly assigned the TERM_EDGE property. Without the assignment, data and control nets may cross the whole processor, producing inefficient layouts. After assigning the TERM_EDGE property, most nets become local and the layout is much cleaner. Figure 2.23 shows the comparison of two layouts with and without assigning

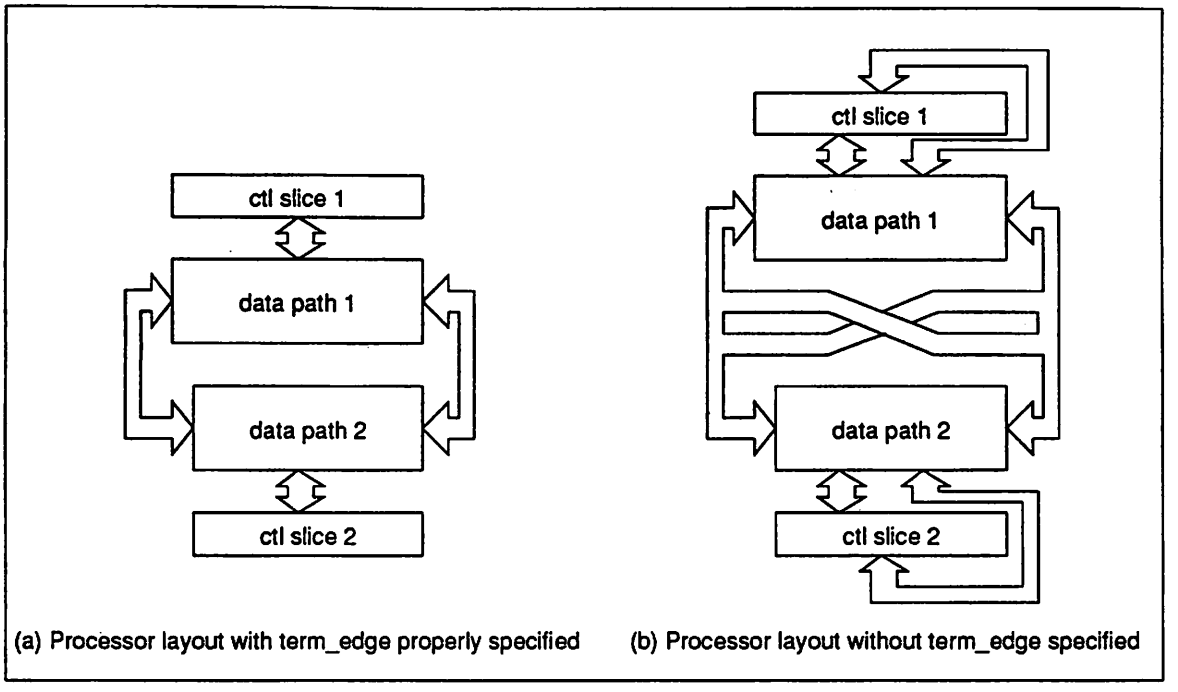

Figure 2.23: Comparison Between Layouts With and Without TERM_EDGE Property

this property. It's obvious that the layout with the TERM_EDGE property assigned is much cleaner than the other one. Notice that after assigning the TERM_EDGE property, the data path sizes may increase if not enough feed throughs are provided by the data path modules. Even with the possible area increase in the data paths, the overall processor area reduces due to a more efficient processor routing.

For the control terminals of the data path modules, the TERM_EDGE property is easily decided from the hardware database. For the data terminals, however, deciding the TERM_EDGE property optimally at this stage is difficult since the placement of the data path blocks won't be performed until the silicon compilation step. Currently, the TERM_EDGE property of the data terminals is assigned based on two constraints. First, both sides of the data paths have approximately equal amount of data nets to avoid routing bottleneck. Second, the TERM_EDGE property is assigned so that all the terminals within one net are located at the same side of the data paths.

More sophisticated heuristics have been tested to improve the layout quality. One of the heuristics is to avoid multiple-way nets in the data path partitioning to simplify the routing problem. Another improvement of the hardware mapper is to allow user interaction

so that the number of data paths or the maximal length of a data path can be specified by the designer. These options allow more freedom to the users in exploring the design space based on the characteristics of each individual design.

2.10 Conclusion on Hardware Mapping

In this chapter, the hardware mapper of HYPER is described in detail including the input format, the target architecture, and the major synthesis process. The hardware database, which provides the necessary area and timing information of all the hardware modules, is also discussed. Performance of the hardware mapper has been demonstrated through some real applications and several modifications have been made on the mapper to improve the layout quality. Future extension of the hardware mapper involves the following subjects:

- A testing strategy as part of the database. This strategy requires the operational information of the normal mode and the testing mode as well as the interconnect information of the scan path. Based on the industrial standard, JTAG can be chosen as the starting point to implement the testing strategy.

- Various clocking strategies. The current clocking strategy of HYPER is based on the two-phase non-overlapped clock. Designs of one-phase clock or more sophisticated clocks such as the four-phase clock should also be synthesizable from HYPER. To do this, additional information for various clocking strategies should be provided for each hardware modules. That is, the CTL-IN-TERMINAL attributes in the hardware database have to be changed. In addition, the hardware mapper needs to be modified so that the control signals are triggered at appropriate phases and the clock nets are properly routed.

- Other hardware platforms. The current target architecture of HYPER is the bit-sliced data-path clusters. Other hardware platforms such as standard cells or gate arrays may be appropriate for certain applications. To accommodate these different hardware platforms, the transformation process of the hardware mapper should be made flexible so that the transformations can be performed optionally in different orders. Furthermore, more transformations should be introduced for the new hardware platforms to achieve high quality designs.

- More precise area estimation of the final layout. The current hardware mapper provides users with the number of nets and the area of each module for area estimation. A more precise estimation can be made based on the HYPER hardware model and the routing efficiency extracted from real layouts. This estimation can assist designers in determining the design quality without going through the layout generation phase.

Chapter 3

Hardware Module Selection

3.1 Motivation and Problem Definition

Given a behavior description of an algorithm represented by a signal flow graph, the goal of hardware selection is to select a proper clock period (if not specified by the user), to link every operator in the flow graph to a hardware library element, and to cluster operations into groups (called *composite* nodes), so that a minimal hardware cost is obtained under timing and throughput constraints.

The motivation of the hardware selection and clustering is illustrated using Figure 3.1. A hardware database and a flow graph are given at the top of this figure. The database contains a barrel shifter (BS) to perform shift operations (>>) and two adder circuits, the carry ripple adder (CRA) and the carry select adder (CSA), for additions (+). The database also contains information on the area and delay of these operators. Consider the given flow graph and assume that the clock rate is 35 nsec, two possible implementations are shown at the bottom of Figure 3.1. The first implementation uses a fully pipelined structure (i.e. all intermediate variables are stored in registers), while the second implementation tries to cluster operations into composite nodes. Although Implementation 1 uses a cheaper adder (CRA), it probably will require more registers than Implementation 2. Furthermore, Implementation 1 takes three clock cycles to process a sample, while Implementation 2 only needs two clock cycles. The increased latency of Implementation 1 might not matter in pipelinable algorithms, but is a major bottleneck in recursive algorithms, which form the majority of the real-time systems. In light of this example, the module selection and clustering process can be rephrased as selecting sets of execution units, which may be

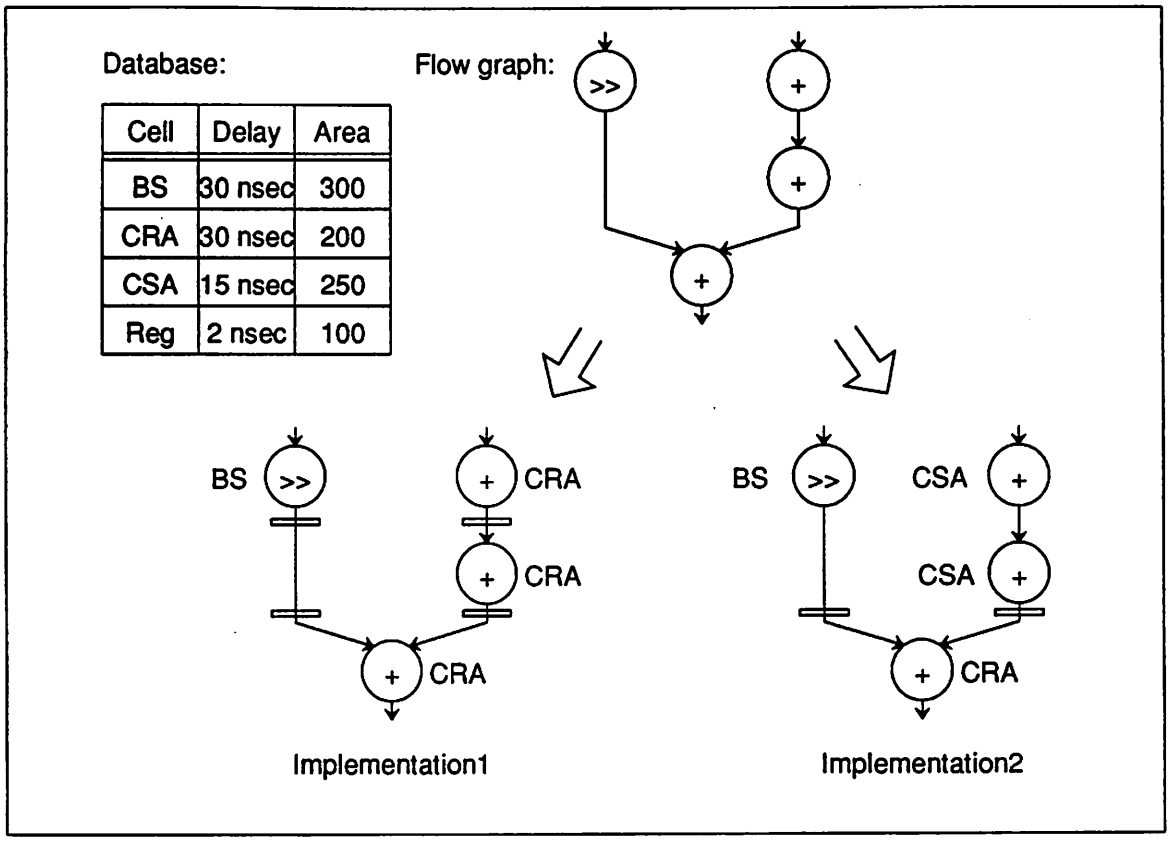

Figure 3.1: Illustration of the Motivation

composite, to minimize the area cost under timing and throughput constraints.

The hardware selection process, as described above, is a non-trivial task. It consists of the following elements: a search strategy, a hardware cost estimation, and a timing model for accurate timing analysis. The relationship of these elements are that the hardware cost estimation directs the search, while each proposed solution is checked with the timing analysis.

In the HYPER synthesis environment, hardware module selection is performed before estimation, hardware allocation, assignment, and scheduling. Therefore, very little information is available at this stage. One of the major challenges for the hardware selector is to come up with a reasonable cost function with limited information. Several cost functions have been experimented and a relaxed-scheduling approach is finally chosen to estimate the cost. The timing model is another important issue. It has to be efficient and capable of capturing the delay properties of all types of function nodes. Finally, the search of the

optimal solution is a NP-hard problem [66]. Heuristics have been developed to perform an efficient search.

3.2 Existing Hardware Selection Approaches

A survey of the previous efforts in module selection can be found in [66]. The approaches that have been proposed can be put into three categories: employing the mixed integer linear programming (MILP) or the integer linear programming (ILP) techniques [2] [10] [40], performing local optimization based on a goodness measure or a rule-based system [45] [47] [57] [11], and finding an optimal solution of a simplified problem (for instance, pipelined design) [36] [66].

Comparing with the existing approaches, the HYPER hardware selection has the following features:

- The algorithm is able to handle recursive, hierarchical, real-time graphs with timing constraints.
- A ripple timing model, which accurately models the timing behavior of each hardware module at the block level, is proposed. Since HYPER allows multiple clock-cycle operations and operation chaining to achieve an efficient design, the proposed timing model has to accurately represent the module delay. Furthermore, the timing analysis has to be efficient since the analysis has to be performed for each intermediate solution in the search process,
- An efficient search mechanism based on clustering is developed. It is well known that the problem of module selection is combinatorial in nature. Therefore, an efficient algorithm is a must.
- An accurate hardware cost estimation is employed. As described above, hardware selection takes place before allocation, assignment, scheduling and all the other synthesis steps in HYPER. The advantage of having hardware selection before the rest of the synthesis steps is that more information about the delay and area of hardware modules is available to the scheduler, resulting in more efficient designs [57]. However, this arrangement makes the cost estimation very difficult during hardware selection.

A hardware cost estimation, which is based on a relaxed-scheduling technique and reflects not only the execution unit cost but also the register cost, is therefore used.

Section 3 describes the proposed approach in detail. After a global description of the algorithm, a detailed description of each sub-task is given in the following sections. Some benchmark results are presented followed this description. Finally, conclusions are drawn and future developments are discussed.

3.3 Clustering Based Module Selection

The proposed technique is based on an iterative node clustering/declustering strategy. Driven by an overall search, nodes are clustered and hardware modules are selected such that the overall cost is minimized. Each proposed solution is checked against the timing constraints using a simple, yet accurate timing model. The algorithm for clustering and hardware selection can be summarized as follows:

```
read flow graph description and hardware database;
feasibility test:
    allocate fastest hardware;
    if (not meet throughput constraint) return FAIL;
for each possible clock rate, do {
    initialization:
        allocate cheapest hardware;
        assign each variable to registers;
    until meeting stop criteria, do {
        order clustering/declustering candidates using similarity test;
        pick one candidate probabilistically;
        perform clustering/declustering;
        (swap in more expensive hardware if necessary)
        compare cost and update flow graph;
    }
}
update flow graph according to the best solution found;
```

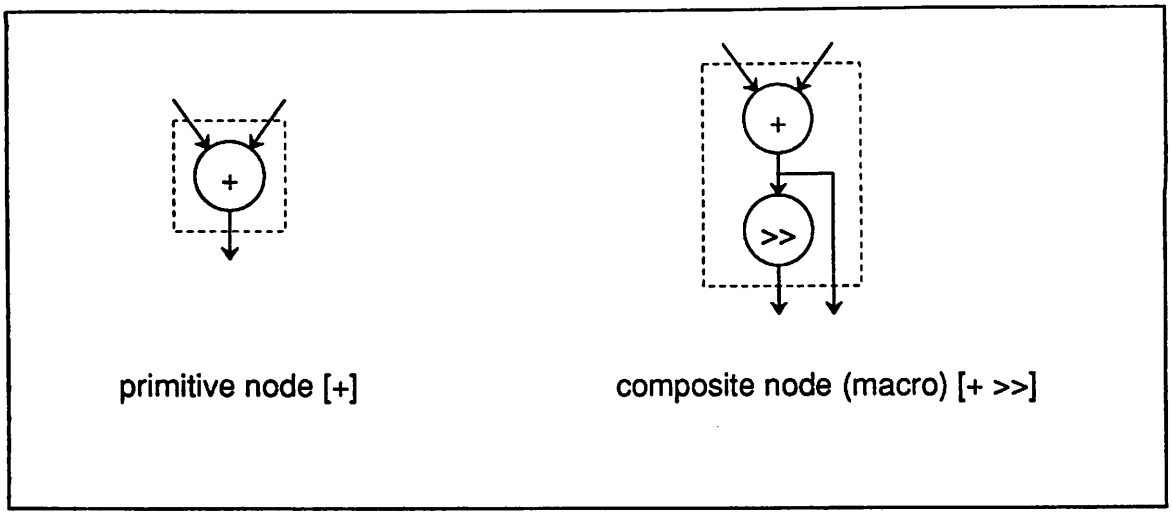

Figure 3.2: Primitive Node and Composite Node

Before going into further detail of the algorithm, two terms are defined. We call a cluster a *primitive node* if there is only one flow graph node in the cluster. If there is more than one node in a cluster, this cluster becomes a *composite node* (or a *macro node*). We use the notation [op1 op2 ... opN] to represent a composite node with N operations, op1, op2, ..., and opN. Figure 3.2 shows a simple example of a primitive node [+] and a composite node [+ >>].

3.4 Possible Clock Rate

If the clock rate is not specified by the user, the algorithm will decide on an optimal rate by scanning over the possible solutions and selecting the one with the lowest cost. Given the throughput constraint (T nsec) of an algorithm, the possible clock periods for the algorithm are T nsec (one clock cycle per sample), T/2 nsec (two clock cycles per sample), T/3 nsec etc. The lower bound of the clock period is limited by the module delay of the fastest possible module selected for the algorithm. That is, $min\_clock\_period = T/n \geq min(module\_delay)$. Notice that lower clock rates are always easier to implement. For this reason, it is advantageous to avoid very small clock periods, even though the proposed approach has no problems in handling graphs where most of the operations take multiple cycles.

3.5 Timing Analysis

During the clustering process, the delay of each proposed cluster has to be checked against the available clock-period. A fully expanded bit-level model has been proposed [40]. This model is very accurate indeed, but is too time-consuming. On the other hand, a straightforward timing model, which approximates the timing delay as the sum of the delays of the composing modules, is used in other synthesis systems. This method is very fast, but does not accurately model the behavior of a chain of operators. In our approach, the timing analysis has to be performed repeatedly during the clustering process. HYPER therefore uses an accurate, yet easily computable *ripple model* to simplify the timing estimation problem.

The model is based on a number of observations, demonstrated in Figure 3.3. The critical path of a Carry Ripple Adder (CRA) is $M1 + N1$, where $M1$ and $N1$ are the one-bit delay and the ripple delay of the CRA. When two CRA's are concatenated, the critical path becomes $2 * M1 + N1$. Notice that the ripple delay $N1$ doesn't double in this case. In the third example where we have one CRA followed by a comparator, the critical path is $N1 + M1 + N2 + M2$. Both ripple delays are included in the critical path due to the fact that the CRA and the comparator have different ripple directions. The last example shows a more complex case where a shifter is located between two carry ripple adders. Although the shifter doesn't have a ripple delay, it causes a *ripple offset.* Therefore, the critical path should also include a partial ripple delay of the last carry ripple adder.

The ripple model characterizes a hardware block by three parameters: the ripple delay (RD), the one-bit delay (OBD), and the ripple offset (RO). The ripple delay expresses the propagation delay of a module as a function of its hardware parameters such as the word length. A positive RD implies that the ripple direction is ripple-to-the-left (LSB to MSB) and a negative RD implies that the ripple direction is ripple-to-the-right (MSB to LSB). This function can be any expression (e.g. linear, log, or square root) to capture the delay behavior of execution units such as carry look ahead adders and carry select adders. The one-bit delay is the critical delay of the one-bit operation. It can also be a function of the hardware module parameters. The ripple offset is used for some special modules such as shifters which cause a disruption in the critical path (without actually having any internal inter-bit ripple). It is also a function of the hardware parameters and can be either positive or negative to represent the shift directions.

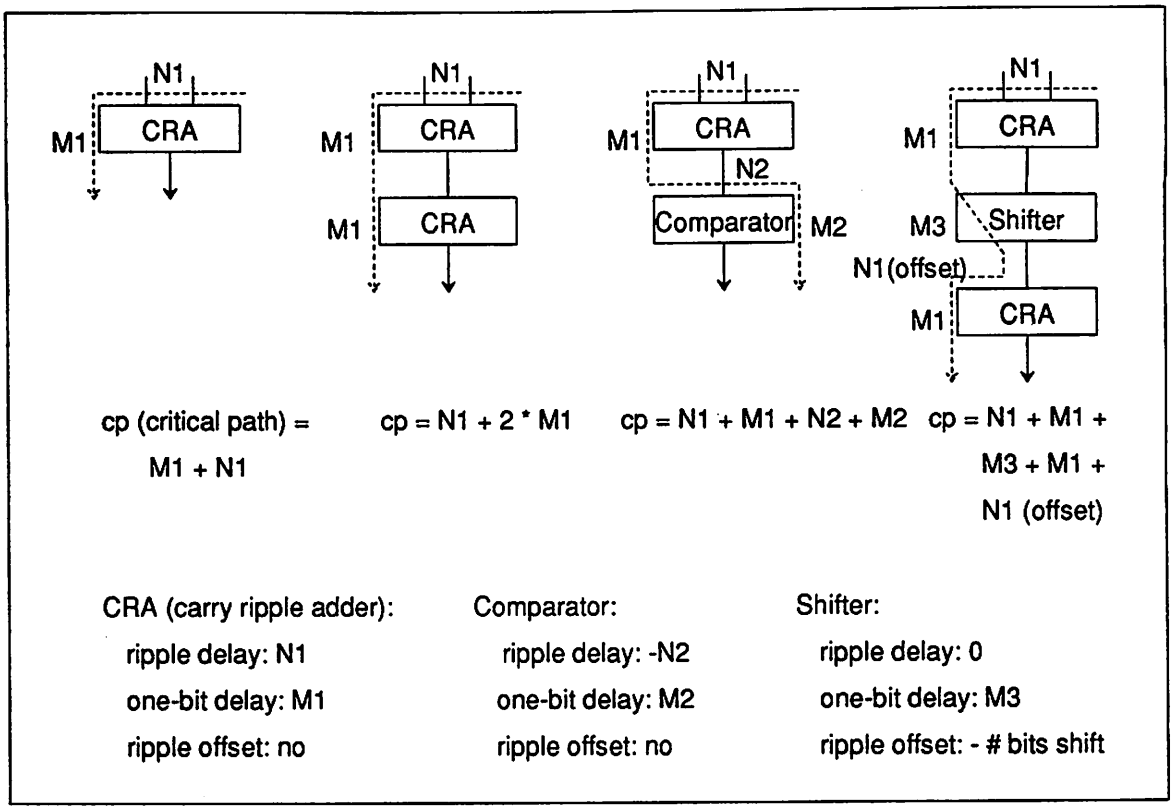

Figure 3.3: Operation Chaining

The critical path of a flow graph is then estimated by tracing the graph and maintaining three parameter values for each edge – the longest ripple delay (LRD) so far (including the module associated with the delay), the total accumulated delay (AD), and the current ripple offset (CRO). These values of an edge can be derived from the ripple parameters of its input node and the parameters of the input edges of the node. The derivation starts with all the input edges initialized to LRD = 0, AD = 0, and CRO = 0. If a register is assigned to an edge, this edge can be considered as an input edge. For each node, we consult the hardware database to calculate its ripple parameters. With the flow graph topologically ordered, we proceed to derive the critical path of the flow graph using some basic derivation rules, which will be given in the next section. Figure 3.4 shows a simple example to demonstrate how the ripple model finds the critical path of a flow graph. For each node in this figure, the three numbers represent the RD, the OBD, and the RO of the node respectively. Similarly, each edge is labeled and annotated with the LRD, the AD, and the CRO. Consider, for example, the chained [+ >> +] operation. The shifter causes a

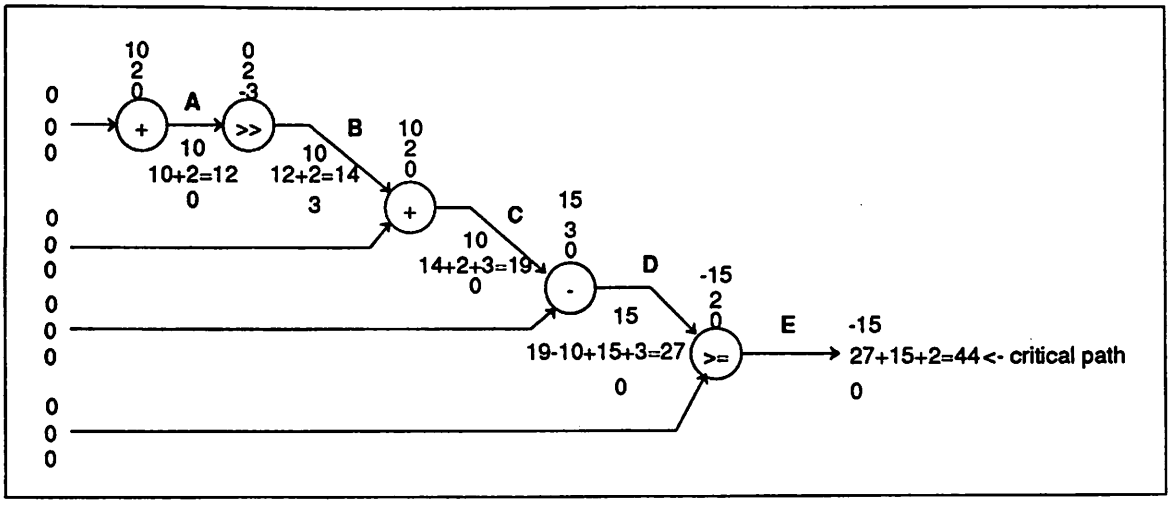

Figure 3.4: Flow Graph Example to Demonstrate the Derivation Rules

ripple offset and therefore a partial ripple is included in the AD of Edge C. To calculate this value, backtracking is required since two cases are possible: (a) The critical path includes the whole ripple of the first addition and the partial ripple of the second addition. (b) The critical path includes the partial ripple of the first addition and the whole ripple of the second addition.

Consider another example: the chained addition and subtraction in the figure. The addition and the subtraction have the same ripple direction, but the subtraction has a longer RD. According to the derivation rules, the LRD after the subtraction should take the longest ripple so far and therefore the RD of the subtraction. Furthermore, the AD should include the RD of the subtraction, but not the RD of the addition. Therefore, the LRD is:

```
LRD after addition - RD of addition + RD of subtraction + OBD of subtraction
= 19 - 10 + 15 + 3
= 27
```

Consider the last example: the chained subtraction and comparison (>=). These two operations have different ripple directions; therefore, the AD and thus the critical path should include the RD's of both operations. The ripple direction and LRD of the output edge are modified according to the >= node. The critical path based on the derivation is:

```
LRD after subtraction + RD of comparison + OBD of comparison
```

= 27 + 15 + 2

= 44

The advantage of the proposed approach is that the modeling is at the module level, resulting in a fast and module-width independent analysis. Even with the possible backtracking, the analysis can be performed efficiently. On average, the timing analysis takes less than 1% of the total run time. Moreover, this model is very accurate in the sense that it correctly reflects the ripple characteristics of hardware modules. Although false timing path problems can still occur, most of the problems are avoided by incorporating the false paths into the model (for instance, a carry bypass adder). Finally, the ripple model is also very general. As described before, the ripple delay, the one-bit delay, and the ripple offset can all be functions of module parameters. We can therefore easily characterize all types of delay behavior using this model. More precise modeling at this level of abstraction does not make sense since the effects of wiring and loading can only be computed when the final architecture is known.

3.5.1 Derivation Rules

The three ripple parameters of an edge can be derived by the ripple parameters of its input node and the ripple parameters of the input edges of that node. In this section, the derivation rules of the ripple model will be given. The best way to analyze different ripple cases is by looking at the ripple directions (RDir) of the input node and its input edges and then comparing the AD and LRD of the edges. We will start the analysis without ripple offset first and extend the analysis to include the cases with ripple offset.

No Ripple Offset

Assume that the input edges have no ripple offset and that the input edges are data edges (i.e. they are not control lines.), the timing analysis includes three different cases as will be described below. To demonstrate the different cases, we assume the model shown in Figure 3.5. In this model, Node C has only two input edges, A and B. The order of A and B doesn't affect the derivation rules and it is straightforward to generalize the derivation with more input edges. The three ripple parameters of Edge D have to be computed.

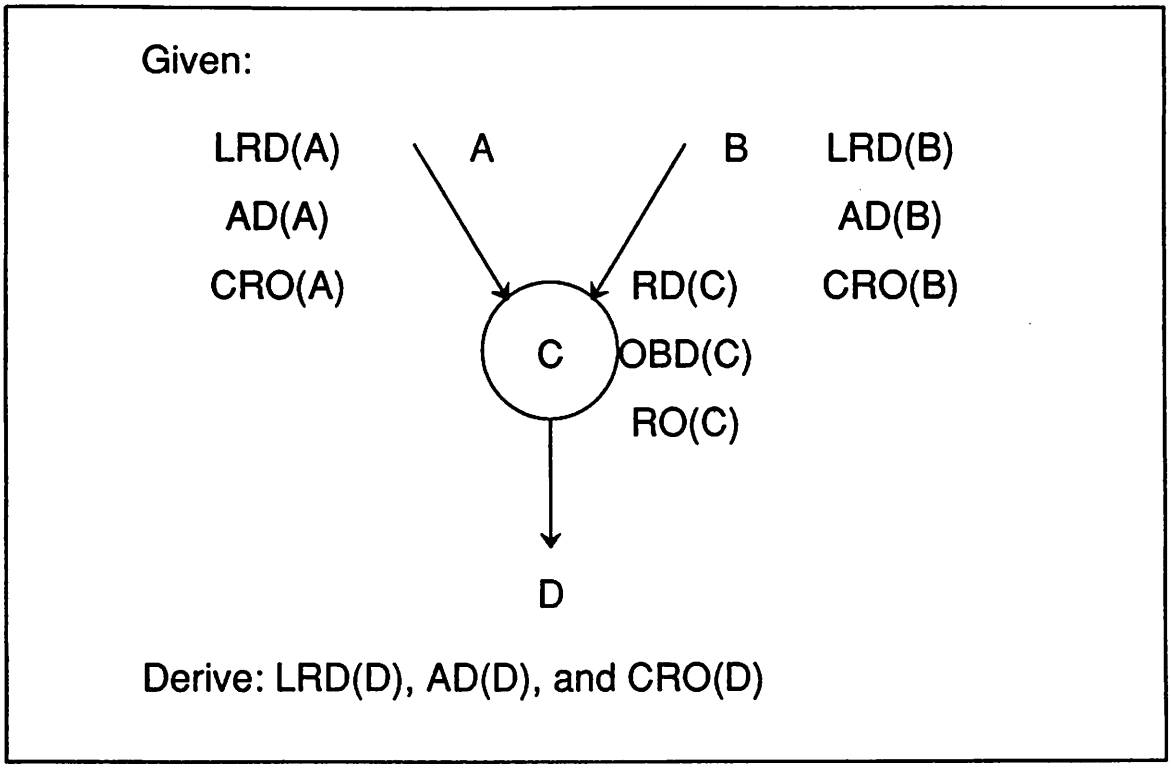

Figure 3.5: Flow Graph to Demonstrate the Derivation Rules

case 1: RD(C) = 0 In this case, Node C has no ripple delay. The ripple parameters of D depends on the bigger value of AD(A) and AD(B)[1]. Suppose that $AD(A) >= AD(B)$, the derivation rules are:

```
LRD(D) = LRD(A)
AD(D) = AD(A) + OBD(C)
CRO(D) = RO(C)
```

case 2: RD(C) > 0 In this case, Node C ripples to the left such as an adder. There are several possibilities and the ripple parameters of D depend on the *maximal AD values* of these possibilities:

1. For input edges that have zero-valued LRD's (e.g. LRD(A) = 0), we have:

   ```
   LRD(D) = RD(C)
   ```

[1]Since the signs of these values represent the ripple directions, we are only comparing the absolute values of AD(A) and AD(B).

```
AD(D) = AD(A) + OBD(C) + RD(C)
CRO(D) = RO(C)
```

2. For input edges that have positive LRD's (e.g. LRD(A) > 0), we have:

```
LRD(D) = max(LRD(A), RD(C))
AD(D) = max(AD(A)+OBD(C), AD(A)-LRD(A)+RD(C)+OBD(C))
CRO(D) = RO(C)
```

3. For input edges that have negative LRD's (e.g. LRD(A) < 0), we have:

```
LRD(D) = RD(C)
AD(D) = AD(A) + RD(C) + OBD(C)
CRO(D) = RO(C)
```

case 3: RD(C) < 0 In this case, Node C ripples to the right such as a comparator. The derivation rules are similar to case 2 except the signs of the parameters, which indicate the ripple directions. For clarity, the rules are listed below. Notice that $RD(C) < 0$ in this case.

1. For input edges that have zero-valued LRD's (e.g. LRD(A) = 0), we have:

```
LRD(D) = RD(C)
AD(D) = AD(A) + OBD(C) - RD(C)
CRO(D) = RO(C)
```

2. For input edges that have negative LRD's (e.g. LRD(A) < 0), we have: (Notice that the maxes in the following formulas are taken on the *absolute values.*)

```
LRD(D) = max(LRD(A), RD(C))
AD(D) = max(AD(A)+OBD(C), AD(A)+LRD(A)-RD(C)+OBD(C))
CRO(D) = RO(C)
```

3. For input edges that have positive LRD's (e.g. LRD(A) > 0), we have:

```
LRD(D) = RD(C)
AD(D) = AD(A) - RD(C) + OBD(C)
CRO(D) = RO(C)
```

For control inputs to Node C, their CRO's and LRD's are both 0. The derivation rules as described above still apply. Similarly, if Node C produces control outputs such as

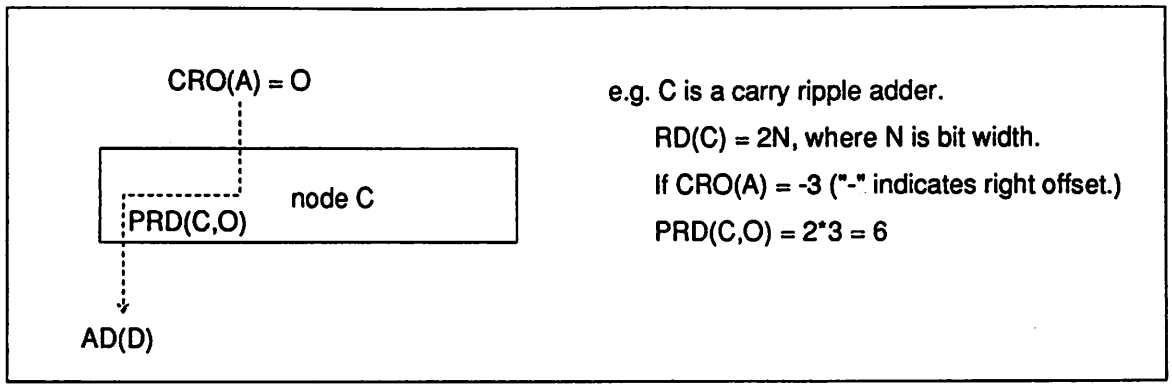

Figure 3.6: Flow Graph to Demonstrate the Ripple Offset

the carry-outs of additions, the output edges will have both its CRO and LRD equal to 0. The AD values can be calculated using the above derivation rules.

Ripple Offset

In this section, the derivation rules are extended to handle the ripple offset cases. Figure 3.5 will still be used as the hardware model and the ripple parameters of Edge D will be derived.

Assume that CRO(A) is not zero, CRO(D) is simply $CRO(A) + RO(C)$. This is due to the accumulative property of ripple offset. CRO will be reset to 0 when a ripple operator such as a carry ripple adder is met. To derive LRD(D) and AD(D), we consider the following three cases:

case 1: RD(C) = 0 Since Node C doesn't have a ripple delay, ripple offset does not affect the derivation rules. The rules described in the previous section still apply.

case 2: RD(C) > 0 Node C ripples to the left such as a carry ripple adder. Before giving the rules, we need to define a new notation PRD (Partial Ripple Delay). $PRD(C,O)$ is the partial ripple delay of Node C due to the ripple offset O. Figure 3.6 illustrates the situation and gives a simple example of the PRD. Having defined PRD, we now consider six possibilities for case 2:

1. Assume that LRD(A) = 0 and that CRO(A) < 0, the ripple offset doesn't affect the derivation:

```
LRD(D) = RD(C)
AD(D) = AD(A) + OBD(C) + RD(C)
```

2. Assume that LRD(A) = 0 and that CRO(A) > 0, the ripple offset shortens the critical path:

```
LRD(D) = RD(C)
AD(D) = AD(A) + OBD(C) + RD(C) - PRD(C, CRO(A))
```

3. Assume that LRD(A) > 0 and that CRO(A) < 0, we have two possible critical paths: (a) The critical path includes whole LRD(A) and partial RD(C). (b) The critical path includes partial LRD(A) and whole RD(C). The derivation rules can be written in the following formula:

```
AD(D) = max(AD(A) + OBD(C) + PRD(C, CRO(A)),
            AD(A) - LRD(A) + RD(C) + OBD(C) + PRD(A, CRO(A)))
LRD(D) = LRD(A) if AD(D) takes the first term in the above formula
         RD(C)  otherwise
```

4. Assume that LRD(A) > 0 and that CRO(A) > 0, there are also two possible critical paths as described in the following formula. Notice that AD(D) may be *smaller* than AD(A) since the MSB's of A are thrown away in this case.

```
AD(D) = max(AD(A) + OBD(C) - PRD(A, CRO(A)),
            AD(A) - LRD(A) + RD(C) + OBD(C) - PRD(C, CRO(A)))
LRD(D) = LRD(A) if AD(D) takes the first term in the above formula
         RD(C)  otherwise
```

5. Assume that LRD(A) < 0 and that CRO(A) < 0, the derivation rules are:

```
LRD(D) = RD(C)
AD(D) = AD(A) + RD(C) + OBD(C) - PRD(A, CRO(A))
```

6. Assume that LRD(A) < 0 and that CRO(A) > 0, the derivation rules are:

```
LRD(D) = RD(C)
AD(D) = AD(A) + RD(C) + OBD(C) - PRD(C, CRO(A))
```

case 3: RD(C) < 0 Node C ripples to the right. The derivation rules are similar to case 2 except the signs of the ripple parameters, which indicate the ripple directions. The rules are listed below:

1. Assume that LRD(A) = 0 that CRO(A) < 0, we have:

```
LRD(D) = RD(C)
AD(D) = AD(A) + OBD(C) - RD(C) - PRD(C, CRO(A))
```

2. Assume that LRD(A) = 0 and that CRO(A) > 0, the ripple offset doesn't affect the derivation:

```
LRD(D) = RD(C)
AD(D) = AD(A) + OBD(C) - RD(C)
```

3. Assume that LRD(A) < 0 and that CRO(A) < 0, there are two possible critical paths:

```
AD(D) = max(AD(A) + OBD(C) - PRD(A, CRO(A)),
            AD(A) + LRD(A) - RD(C) + OBD(C) - PRD(C, CRO(A)))
LRD(D) = LRD(A) if AD(D) takes the first term in the above formula
         RD(C)  otherwise
```

4. Assume that LRD(A) < 0 and that CRO(A) > 0, we have two possible critical paths: (a) The critical path includes whole LRD(A) and partial RD(C). (b) The critical path includes partial LRD(A) and whole RD(C). The derivation rules can be written in the following formula:

```
AD(D) = max(AD(A) + OBD(C) + PRD(C, CRO(A)),
            AD(A) + LRD(A) - RD(C) + OBD(C) + PRD(A, CRO(A)))
LRD(D) = LRD(A) if AD(D) takes the first term in the above formula
         RD(C)  otherwise
```

5. Assume that LRD(A) > 0 and that CRO(A) < 0, the derivation rules are:

```
LRD(D) = RD(C)
AD(D) = AD(A) - RD(C) + OBD(C) - PRD(C, CRO(A))
```

6. Assume that LRD(A) > 0 and that CRO(A) > 0, the derivation rules are:

```
LRD(D) = RD(C)
AD(D) = AD(A) - RD(C) + OBD(C) - PRD(A, CRO(A))
```

For the control input cases, since CRO's of control inputs are equal to zero, the derivation rules as described in the previous section still apply. For the control output

cases, both CRO's and LRD's of control outputs are equal to 0 and the AD values can be calculated using the rules described in this section.

3.6 Clustering Based Search Algorithm

While clustering can improve clock cycle utilization and reduce register costs, it may also reduce resource sharing and hence increase hardware costs. To address these contradictory requirements, the module selection process is organized as a probabilistic iterative-improvement process. The basic moves are either the clustering of two nodes (primitive or composite) into a composite one or its inverse – the declustering move, which decomposes a composite node into two nodes. The second move is essential when local minima are to be avoided.

For each move, it is first decided whether to perform clustering or declustering. This decision is based on the number of available candidates for each. The number of candidates for clustering is the number of edges, while the number of candidates for declustering is the number of composite nodes. If clustering is chosen, a *similarity test* is performed to measure the possible benefit of clustering for each candidate. Figure 3.7 demonstrates the process through a simple example. In this Figure, the number of clustering candidates and the number of declustering candidates are 4 and 2 (2 composite nodes [++]) respectively. Assume that clustering is probabilistically chosen, the similarity test is then performed. For each edge, a tuple is constructed based on the functionality of its input and output nodes. A table is used to record the occurrence of each instance. In this example, the combination [++] [+] occurs twice in the graph, while other combinations never occur or occur only once. Using the occurrence table as a measure, the algorithm probabilistically selects the candidate for clustering. If, for instance, the [++] [+] combination is selected, the flow graph is updated and the composite nodes [+ + +] are constructed. The flow graph after the execution of this move is shown at the right side of Figure 3.7.

On the other hand, if declustering is chosen as the next move, a different measure is used for calculating the possible benefit of the candidates for declustering. The node with *the least* occurrence is the most probable candidate for declustering.

Graph cycles may be introduced during the clustering process. Figure 3.8 illustrates the situation. If the two nodes to be clustered are respectively the input node and the output node of one single node, graph cycles will be created by clustering. Graph cycles

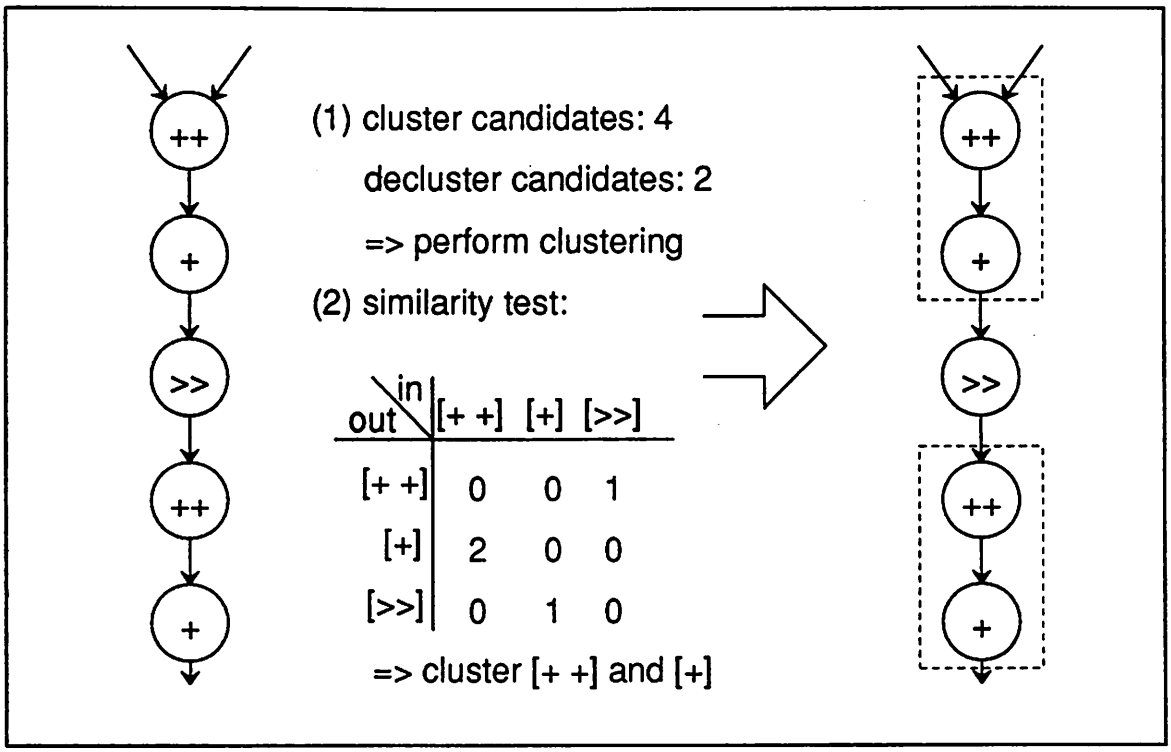

Figure 3.7: Clustering Based Search Strategy

cause a race condition in the flow graph and must be removed to achieve a feasible scheduling. That is, the flow graph must be acyclic. In general, graph cycles will be generated if the two nodes to be clustered have respectively a higher and a lower topological order than a certain node. To avoid the graph cycles, flow graphs are topologically ordered before the clustering is performed. When clustering two nodes, a test, which goes through the nodes with topological orders between the two clustered nodes, is performed to ensure that no graph cycle is produced.

3.7 Hardware Swapping

The hardware selection is performed during the clustering process. Initially, all operations are implemented on the cheapest hardware. During the clustering, more expensive (but faster) hardware might be swapped in if needed to meet the timing constraint. An exhaustive search with pruning is performed to choose the candidate operation for swapping as well as the proper hardware module to execute the operation. This search is inexpensive

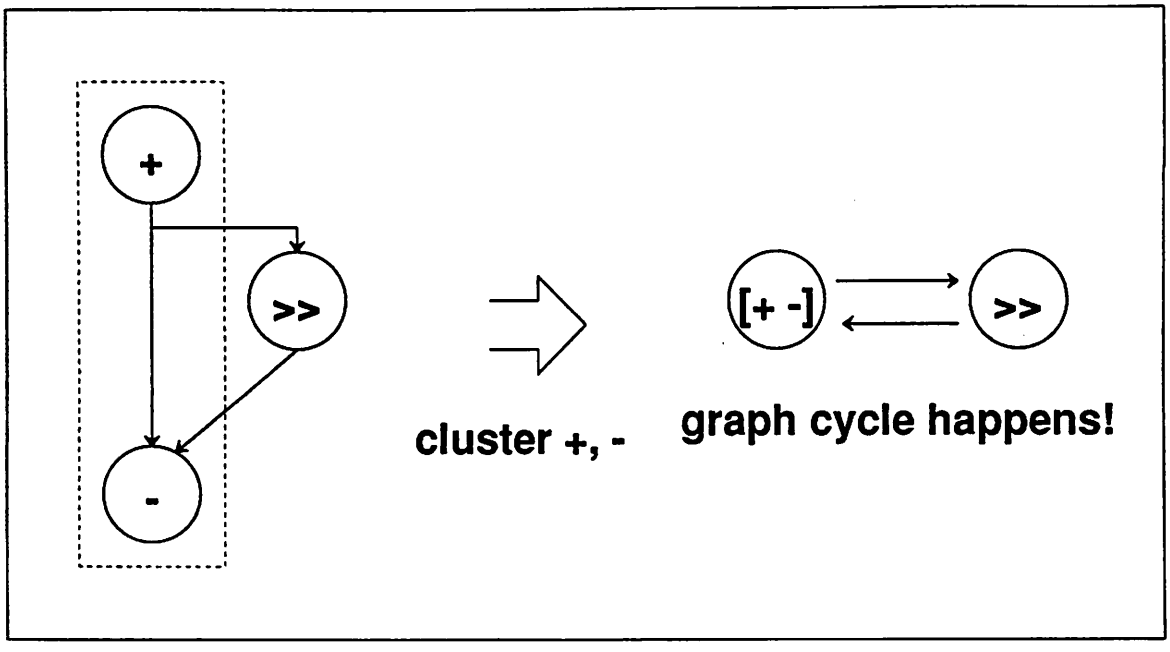

Figure 3.8: Graph Cycles Caused by Clustering

since the choices for swapping are limited for most cases and the pruning can be performed efficiently by evaluating the cost of the proposed hardware function.

Figure 3.9 shows a simple example of the swapping process. To cluster an addition node and a comparison node, the algorithm has to check the timing constraint. Assuming that with the cheapest hardware implementation, the new cluster cannot meet the timing constraint. After searching through the database, a faster module, a carry select adder in this case, is found to replace the cheapest addition module, the carry ripple adder. After the swapping, the new cluster satisfies the timing constraint and the clustering of these two nodes is completed.

An important constraint on composite nodes is introduced to simplify the hardware selection process: *the same implementation is used for all composite nodes with the same flow graph structure* (except the multi-function units). For example, in Figure 3.9, the implementation for the composite node [+ >] is a carry select adder followed by a comparator. From this constraint, all the composite nodes with an addition node followed by a comparison node are implemented by a carry select adder followed by a comparator. This constraint greatly simplifies the hardware selection problem. The hardware selector now only needs to work on the *master* of a composite node instead of each *instantiation* of

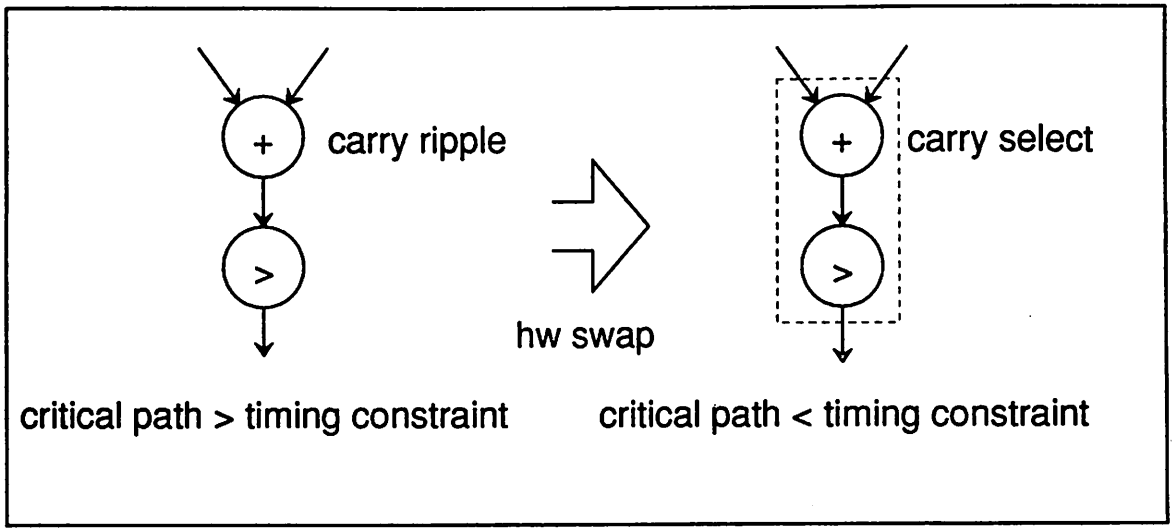

Figure 3.9: Hardware Swapping

the composite node.

The information of the available hardware blocks is provided by the parameterized hardware database system as described in Chapter 2. A set of database access routines have been implemented in the database system to facilitate the hardware selection process.

3.8 Hardware Cost Function

The cost function for the clustering and hardware selection process can be represented by the following formula:

$$Cost = c1 * MAX(0, (CP - t_{max})) + c2 * \sum_{i} (A_{Ri} * N_{Ri})$$

This formula includes two terms which are weighted by $c1$ and $c2$. The first term represents the timing constraint, while the second term represents the hardware cost of the proposed solution. In the first term, t_{max} is the available time of the algorithm given by the designer and CP is the critical path of the algorithm. If the timing constraint is met (i.e. $t_{max} \geq CP$), this cost is 0 and the total cost is equal to the hardware cost; otherwise, the timing cost is more significant than the hardware cost. This implies that $c1$ should be much larger than $c2$. Initially, $c1$ can be small and as the search process moves on, $c1$ will gradually increase and become infinity when the search process ends. This means that an initial

solution may not meet the timing constraint, but as the search process proceeds, the timing constraint becomes more and more important and the final solution must meet the timing constraint.

The second term of the cost function is the estimated total area of the execution units. In this formula, A_{Ri} is the area of resource Ri and N_{Ri} is the number of required Ri's. A_{Ri} can be calculated from the hardware database; however, N_{Ri} can not be exactly determined until after the hardware allocation process. A precise estimation for N_{Ri} is therefore required.

An absolute min-bound can be used to calculate N_{Ri}. This min-bound is computed using the following formula:

$$N_{Ri} \geq \frac{O_{Ri} * D_{Ri}}{t_{max}}$$

Where D_{Ri} is the duration of a single operation of class Ri and O_{Ri} is the number of Ri nodes in the flow graph. A similar bound is also used in SLIMOS [36] and MOSP [66]. Even though it is very simple to calculate, this min-bound is too optimistic and will present a totally wrong picture. It assumes that all operations can be distributed evenly over the available time, which is rarely the case. A more precise bound can be found using a technique called *discrete relaxation* [59]. This approach uses a relaxed-scheduling technique to determine the minimal execution time of the graph given a certain allocation. The relaxation is achieved by considering only one node type at a time and ignoring the precise precedences between nodes. Only the as-soon-as-possible (ASAP) and as-late-as-possible (ALAP) times (as obtained from the graph leveling) are retained. This approach turns an NP-complete problem into a problem of complexity $N_{Ri} * logN_{Ri}$. The overall estimation consists of an iterative procedure, starting from the absolute min-bound. N_{Ri} is increased until a schedule can be found.

Figure 3.10 shows a simple example where the absolute min bound breaks down. Assuming that all operations can be performed within one clock cycle and that the throughput constraint is 3 cycles/sample, the absolute minimum bound estimates that only one adder is needed. However, inspection of the graph clearly shows that at least two adders are needed to meet the throughput constraint due to the distribution of the additions in the earlier time-slots. The relaxed scheduling approach, on the other hand, correctly finds the minimal number of adders needed by the iterative procedure.

Figure 3.11 shows the ratio of the sharp minimum bound obtained by the relaxed

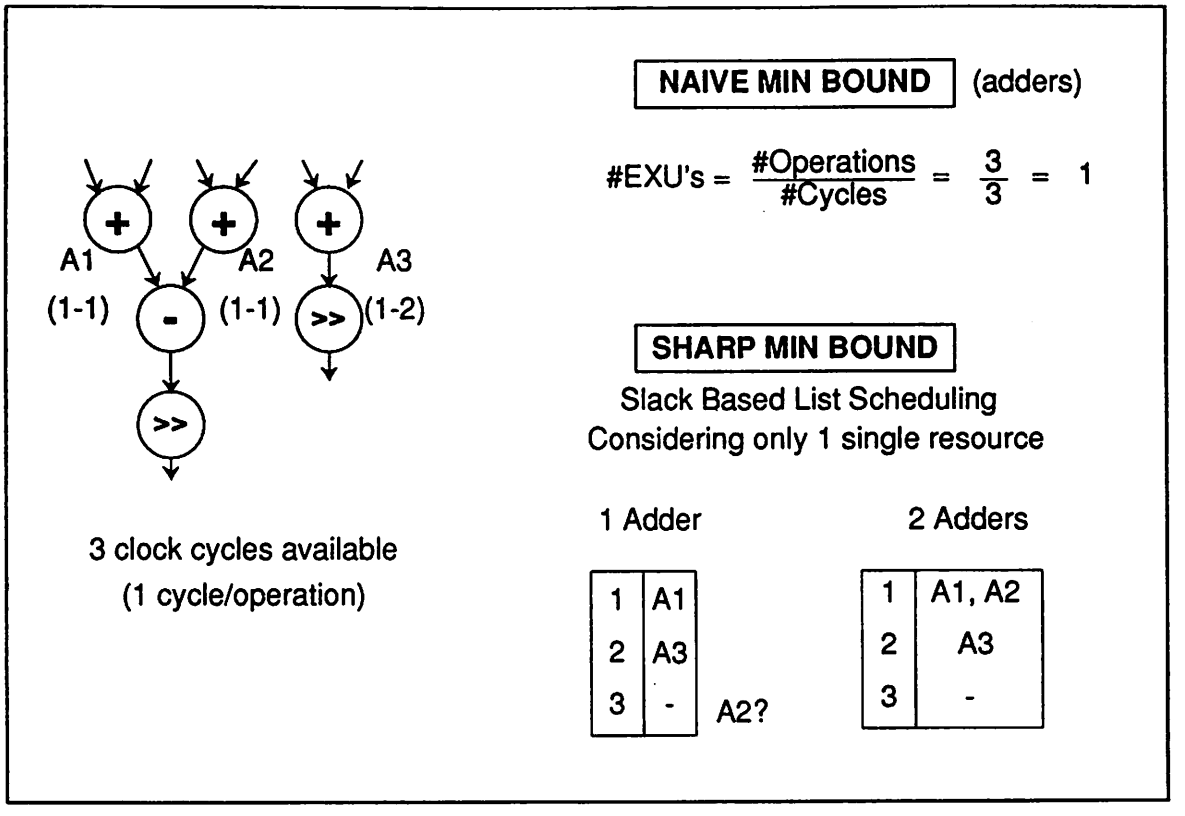

Figure 3.10: Cost Estimation – Min Bound

scheduling technique against the absolute minimum bound for 48 benchmark runs of real applications. An average of 65% improvement on the estimated minimum bound is achieved by the relaxed-scheduling approach. On average, the difference between the sharp minimum bound and the real execution unit cost is less than 15% [62].

The cost of registers is reflected in the tie breaking rules. As described in the algorithm, the search process always keeps the best solution found so far. If a tie happens, two tie breaking rules are used to select the most promising solution. The first rule is to choose the solution with a smaller number of clusters. The reason is that a smaller number of clusters implies a smaller number of variables to be stored in registers and therefore a better chance of finding a solution with less registers. The other tie breaking rule is to choose the solution with a lower clock rate due to a possibly easier implementation.

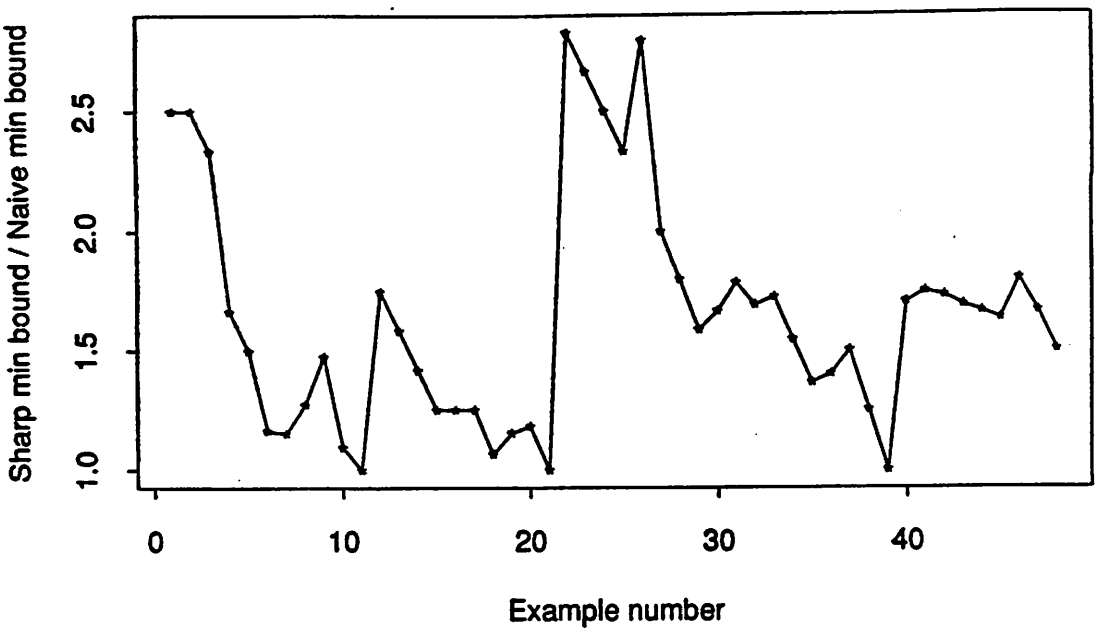

Figure 3.11: Bound Ratio of Execution Units for 48 Examples

3.9 Clustering for Hierarchical Graphs

The proposed approach is capable of performing clustering and hardware selection for hierarchical flow graphs. To facilitate the HYPER synthesis process, all the operation nodes are clustered into several function nodes if there are control macro nodes at the same level of hierarchy. This clustering process is called *functional clustering* and should not be confused with the clustering in the hardware selection and clustering process. Figure 3.12 shows the idea of the functional clustering. In this figure, two operation nodes in front of the *loop* node are clustered into a function node and the last operation node is reduced to the subgraph of another function node. The numbers in italic next to the nodes in the clustered flow graph represent one possible schedule. This functional clustering can be easily performed by a topological ordering.

With the clustering of operations into function nodes, the hardware selection and clustering of hierarchical flow graphs becomes much easier. The clustering and hardware selection process now only needs to be performed on the lowest hierarchy level of the flow graphs where all the operation nodes reside.

Performing the relaxed scheduling for cost estimation on a hierarchical graph is somewhat more complex than the flat graphs since only the total available time is known,

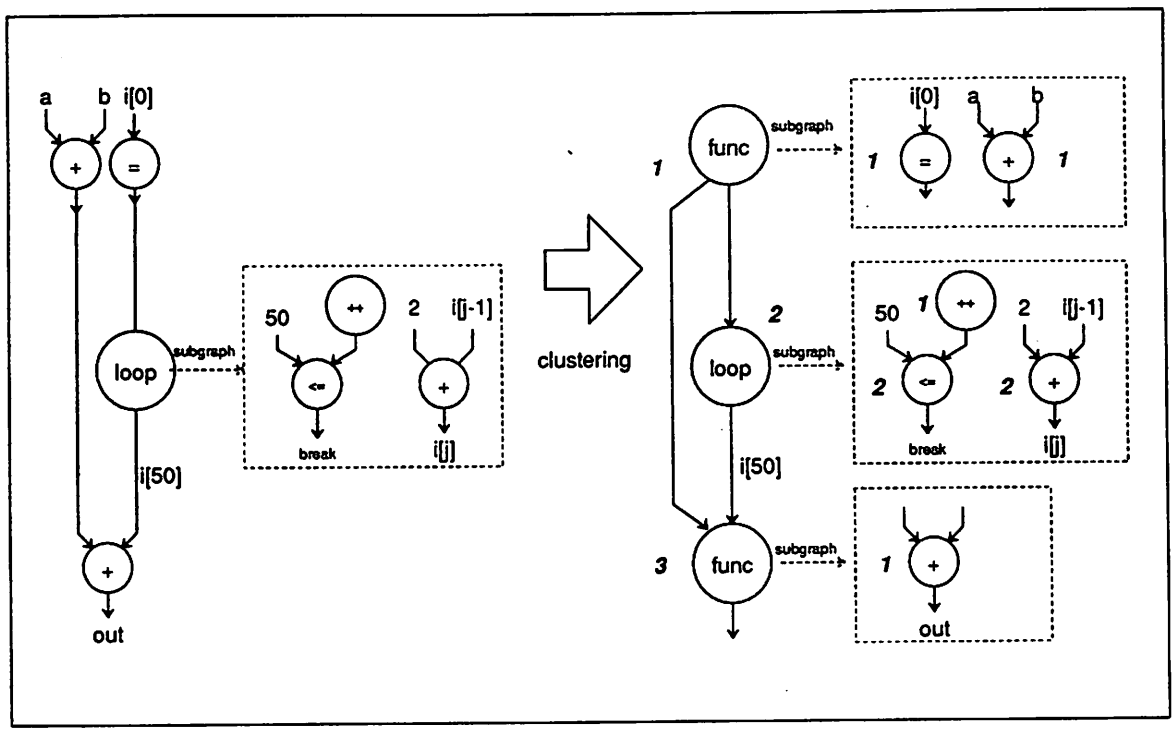

Figure 3.12: Node Clustering in Hierarchical Flow Graphs

but not the time allotted to each subgraph. During the estimation process, we therefore distribute the available time over the subgraphs using a simple heuristic, called the *stress* of a graph, which is defined by the following formula for a graph g:

$$stress(g) = \frac{number\_of\_nodes(g)}{t_{max}(g) - CP(g) + 1}$$

$CP(g)$ represents the critical path of Graph g and $t_{max}(g)$ the current available time. A good time distribution tries to equalize the stress over all subgraphs. A high stress factor implies too many nodes in Graph g, but not enough time is allotted. The time distribution algorithm first sets the available time for each graph to its critical path. Next, more time is allocated to the graph with the highest stress. This process is continued until the available time is completely used. After the time distribution process, the cost estimation can proceed as for the flat graph case.

Figure 3.13 shows a simple example to illustrate the time allotment process. Subgraph 1 has 3 nodes and the critical path is 3 cycles. Subgraph 2 has 4 nodes and the critical path is 2 cycles. Suppose that the total available time is 7 cycles, the extra time is:

$$7 - 3 - 2 = 2cycles$$

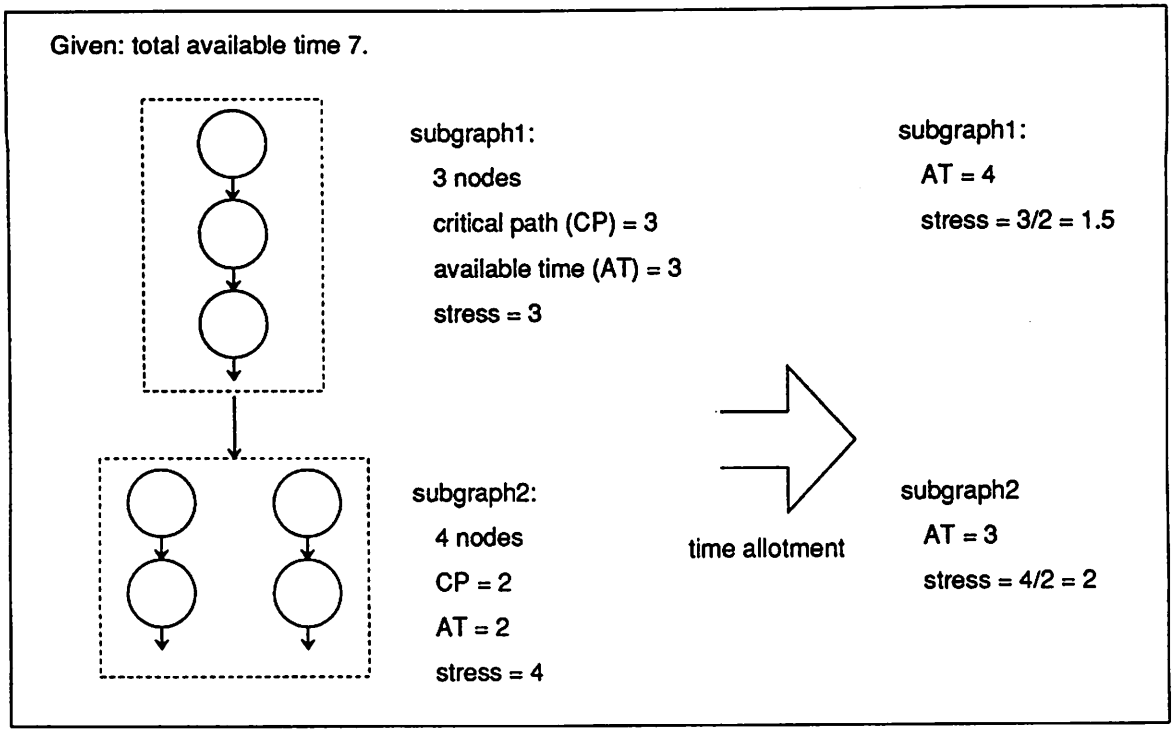

Figure 3.13: Example to Illustrate Time Allotment

This process first calculates the stress for both subgraphs and finds out that Subgraph 2 has a higher stress (4) than Subgraph 1 (3). One extra cycle is therefore allocated to Subgraph 2. With the extra time, Subgraph 2 now has a lower stress (2) than Subgraph 1 (3). Subgraph 1 gets the other extra cycle and the allotment process is completed. The stress of the two subgraphs after the allotment are 1.5 and 2 respectively.

3.10 Experiments and Results

A simple biquad example is illustrated in Figure 3.14. The critical path is marked by a dotted line. Assuming that the designer specifies the sampling rate to be 2MHz. Consider two cases: in case one, the designer specifies the clock rate to be 16MHz, and the throughput constraint is $16/2 = 8 cycles/sample$. With the cheapest hardware and the fully pipelined (i.e. store all variables in registers) structure, the critical path is 8 cycles assuming single cycle operations. The structure satisfies the throughput constraint and the hardware required is two adders, one subtracter, two barrel shifters and fourteen registers. After a similarity test, [>> +] is chosen to be the candidate for clustering. The solution

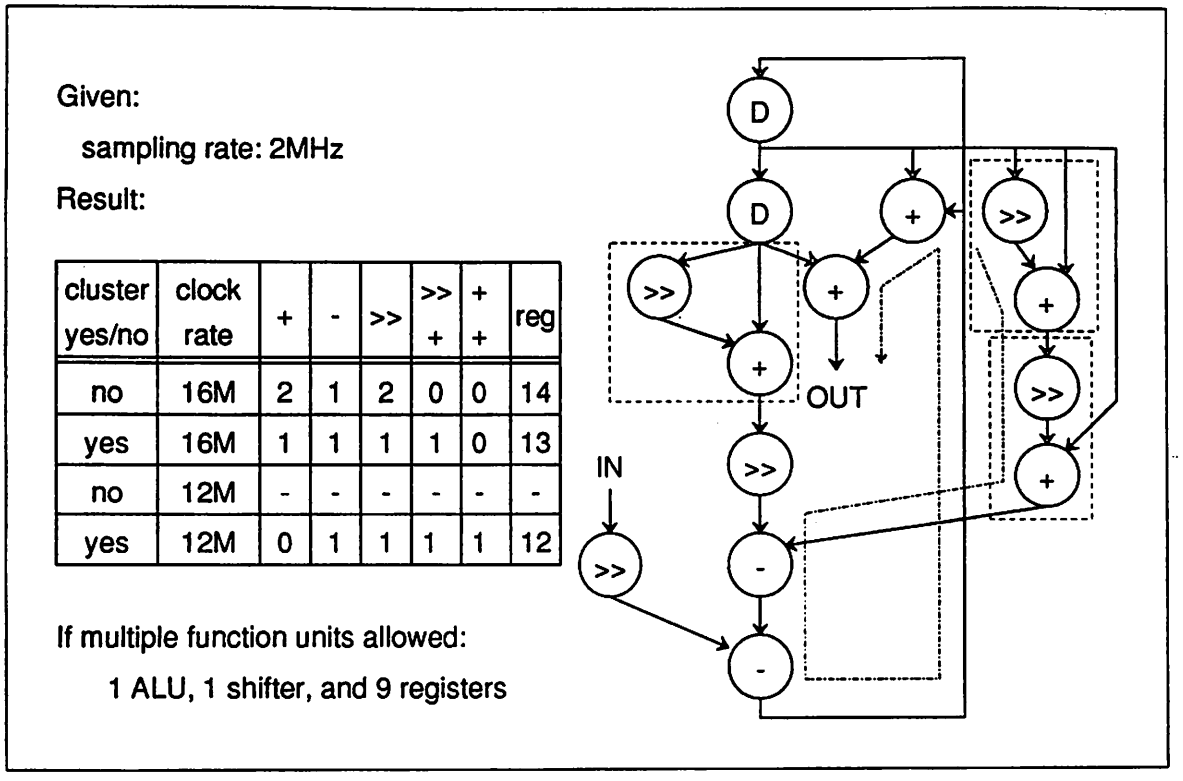

Figure 3.14: A Biquad Example

after clustering is shown with the dotted boxes. The hardware required is shown in the second row of the table in the figure. The cost reduces due to the lower number of required registers. If multi-function units such as an ALU are allowed, this algorithm finds a solution with only one ALU, one shifter, and nine registers. The hardware cost is much lower than those of the solutions with only single-function units. In HYPER, the hardware selector proposes sets of feasible hardware modules including both multi-function units and single function units and the scheduler/allocator can make decisions on which set of modules to use.

Consider the other case in which the designer specifies the clock rate to be 12MHz. Now the fully pipelined structure fails to meet the throughput constraint since the critical path (8) is longer than the cycles allowed (6). After the clustering, the critical path reduces to 6 cycles and meets the throughput constraint. The hardware cost is shown in the last row of the table.

In addition to the probabilistic approach as described in this chapter, a greedy approach which always chooses the most prominent candidate for clustering has also been

benchmark	sample rate	clock (nsec)	time on SPARC	before/ after	#nodes	#edges	#reg.	exu cost
7th order IIR	1 MHz	75	2min	before	45	46	40	164
				after	38	39	35	134
10th order FIR	1 MHz	90	1.6min	before	40	41	33	184
				after	33	34	32	154
5th order WDF	0.7 MHz	100	1.3min	before	34	43	27	460
				after	28	38	27	260
3rd order WDF	1 MHz	70	1.1min	before	25	26	14	72
				after	25	26	10	55
3rd order WDF	2 MHz	60	3.7min	before	25	26	no sol.	no sol.
				after	15	22	12	136

Figure 3.15: Clustering Result

implemented. The benchmark results of the greedy approach along with the results of the probabilistic approach can be used to demonstrate the tradeoff between the time and the performance of these two approaches. For the same biquad example, the greedy approach finds a solution with 1 adder, 1 subtracter, 1 shifter, 2 [>> +] units, and 13 registers. Compared with the probabilistic approach, the hardware cost is one shifter and one register more. The run time is 4 seconds vs. 45 seconds. Notice that two [>> +] units are required in this case. With the absolute minimum bound, one would estimate that only one [>> +] unit is needed. However, through the relaxed-scheduling approach, we correctly estimate that two [>> +] units are required.

Figure 3.15 shows some benchmark results of the probabilistic approach. The clock rates were predefined except the last one in which the algorithm tries to find the best clock rate. The execution-unit costs and the register costs in this table are the real costs after the allocation process. Notice that the register costs cannot be calculated directly from the number of flow graph edges since several variables (i.e. edges) can share the same register. The number of edges, however, gives some indication on the amount of registers needed.

We can see form the results that the clustering process helps to meet the timing

constraints as well as to reduce the hardware costs. For the first three cases, clustering improves the hardware costs. About 83% to 90% of the variables are stored in registers and the register costs are reduced for most cases. In addition, the execution-unit costs reduce after clustering, ranging from 16% to 43%. For the 3rd order WDF case, no clustering is performed since the fully pipelined implementation is very efficient. The reduction of the hardware cost as shown in this table is achieved through the use of multi-function units. For the last case, clustering is required to meet the throughput constraint. This table also shows that all the solutions are found within minutes on a SPARC station.

The same benchmarks are also used to test the greedy approach and the results are shown in Figure 3.16. The performance of the greedy approach is slightly worse than the probabilistic approach for three examples and equivalent to the probabilistic approach for the other two examples. The run time of the greedy approach is much shorter than that of the probabilistic approach. However, the run time of the probabilistic approach is still acceptable. The reason why the greedy approach has a similar performance to the probabilistic approach is that the benchmarks have very regular flow graph structures and therefore the greedy approach can also find good solutions. For irregular flow graphs, the probabilistic approach should perform much better than the greedy approach.

From running the benchmarks and analyzing the results, we observe several interesting points:

- Although multipliers usually are not clustered, the clustered implementation can still save multipliers and greatly reduce hardware costs by clustering other operations. The 5th order WDF as shown in the above table is a good example, in which a multiplier is saved due to the clustering of additions. After the clustering, more clock cycles are available for multiplications and therefore less multipliers are required. Figure 3.17 illustrates the situation.

- *Partial clustering* may be advantageous in some cases. Figure 3.18 shows an example to demonstrate the idea. The first implementation in the figure clusters all instances of [++], while the second implementation only clusters three of the four instances. The second implementation requires less hardware due to a better utilization of the [+] unit (adder). The above situation can usually be detected by a low utilization of some hardware units after clustering. A partial declustering is required in this case to improve the hardware cost.

benchmark	sample rate	clock (nsec)	time on SPARC	prob./ greedy	#nodes	#edges	#reg.	exu cost
3rd order IIR	1 MHz	75	7 sec	prob.	38	39	35	134
				greedy	38	39	37	144
10th order FIR	1 MHz	90	6 sec	prob.	33	34	32	154
				greedy	33	34	32	154
5th order WDF	0.7 MHz	100	8 sec	prob.	28	38	27	260
				greedy	29	39	27	260
3rd order WDF	1 MHz	70	3 sec	prob.	25	26	10	55
				greedy	25	26	14	72
3rd order WDF	2 MHz	60	10 sec	prob.	15	22	12	136
				greedy	14	22	14	156

Figure 3.16: Greedy Approach vs. Probabilistic Approach

- In addition to clustering, we can also try to choose more expensive but faster hardware for multiple-cycle operations to meet the throughput constraint. Figure 3.19 shows a simple example to illustrate the idea. Assume that the available time is 2 clock cycles (40/20 = 2), the slow adder (CRA) takes two clock cycles (35+5 = 40), and the fast adder (CSA) takes one clock cycle (12+5 = 17). The initial solution selects the slow adders for all the three additions. Since clustering the flow graph cannot meet the throughput constraint, hardware swapping is performed. Fast adders are swapped in for the two additions on the critical path and the throughput constraint is satisfied. Hardware swapping can be easily implemented in the proposed algorithm. In addition to the clustering and declustering moves, swapping moves are introduced. The number of candidates for the swapping moves is the number of nodes of multiple-cycle operations. If the swapping moves are probabilistically chosen, the similarity test as described above is performed and hardware is swapped according to the possible benefit of swapping for each candidate.

- Although multi-function units such as an ALU or an adder/subtracter may have a larger area and a longer delay than single-function units, they are more versatile

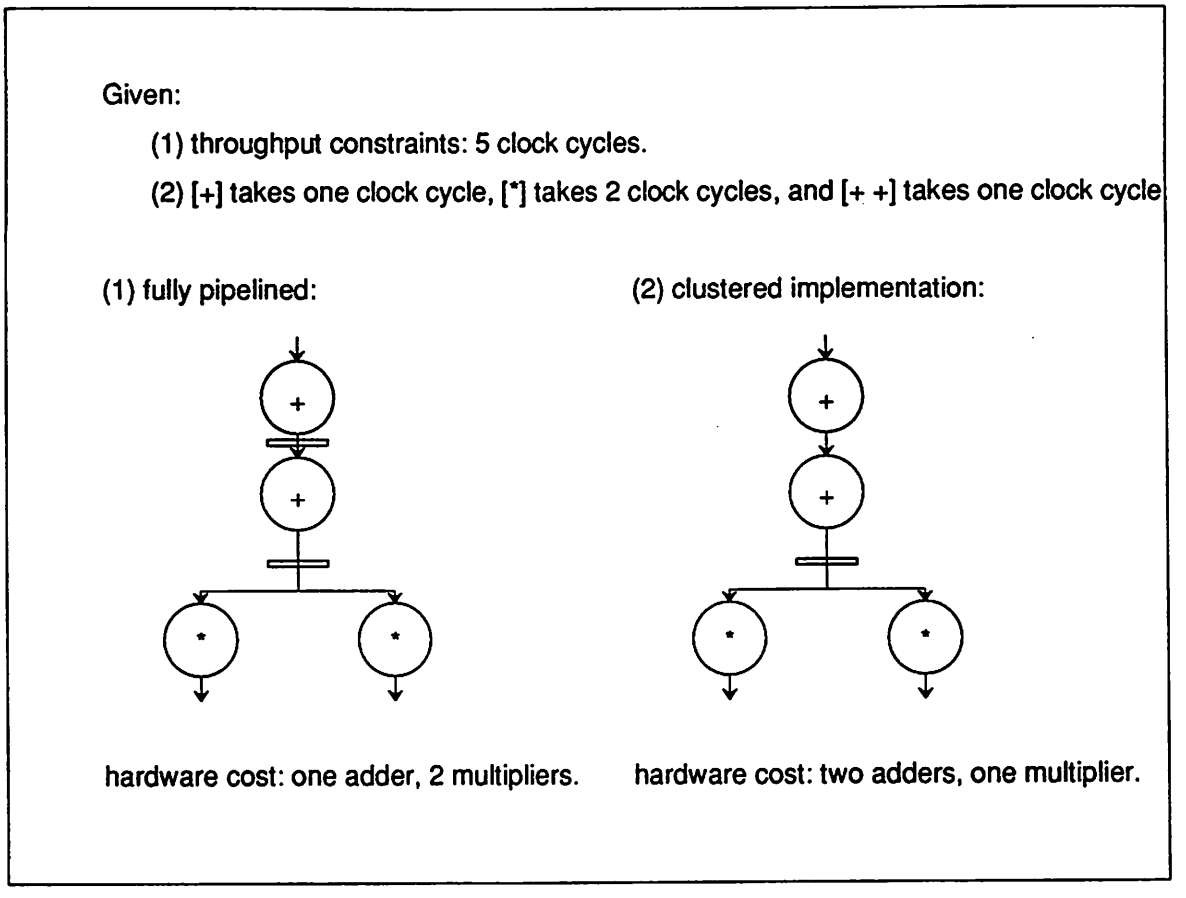

Figure 3.17: Saving Multipliers by Clustering Additions

and therefore offer increased chances for resource sharing. The benchmark results demonstrates that the introduction of multi-function units is extremely important for an efficient implementation.

3.11 Conclusion on Module Selection

An algorithm to perform hardware selection and operation clustering has been developed. This algorithm also decides on a proper clock rate if not specified by the user. The benchmark results show the excellent performance of the proposed algorithm. The contribution of this work can be summarized as follows:

- This algorithm is able to handle recursive graphs with fixed timing constraints.
- Chained operators are allowed based on an accurate timing analysis.

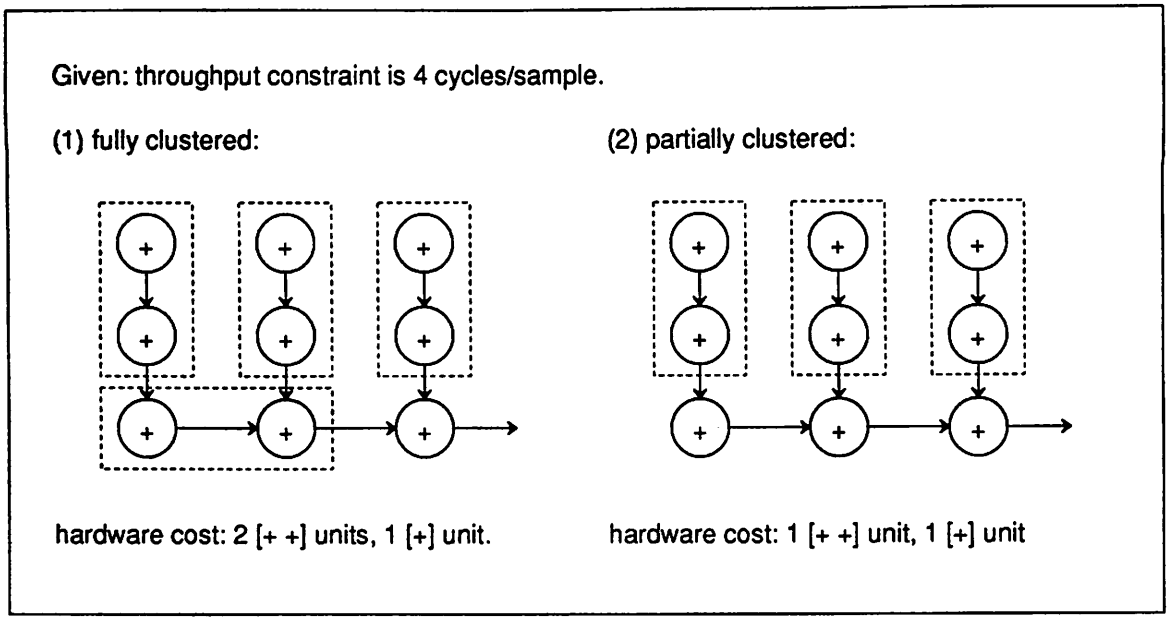

Figure 3.18: Partial Clustering

- A precise, relaxed-scheduling-based cost estimation which considers both the execution-unit cost and the register cost is implemented.
- A clustering-based search strategy efficiently solves the clustering and hardware swapping problem.

Hardware selection and clustering is a relatively new area in high-level synthesis and there is no standard benchmark or standard hardware library to compare with. Moreover, the performance of the hardware selection also depends on how well it interacts with the other synthesis processes such as scheduling, allocation, and transformations. To further prove the quality of the new algorithm, exhaustive search shall be performed on simple examples. For complex examples, layouts should be generated (through the Lager silicon compilation system) to verify the quality of the solutions.

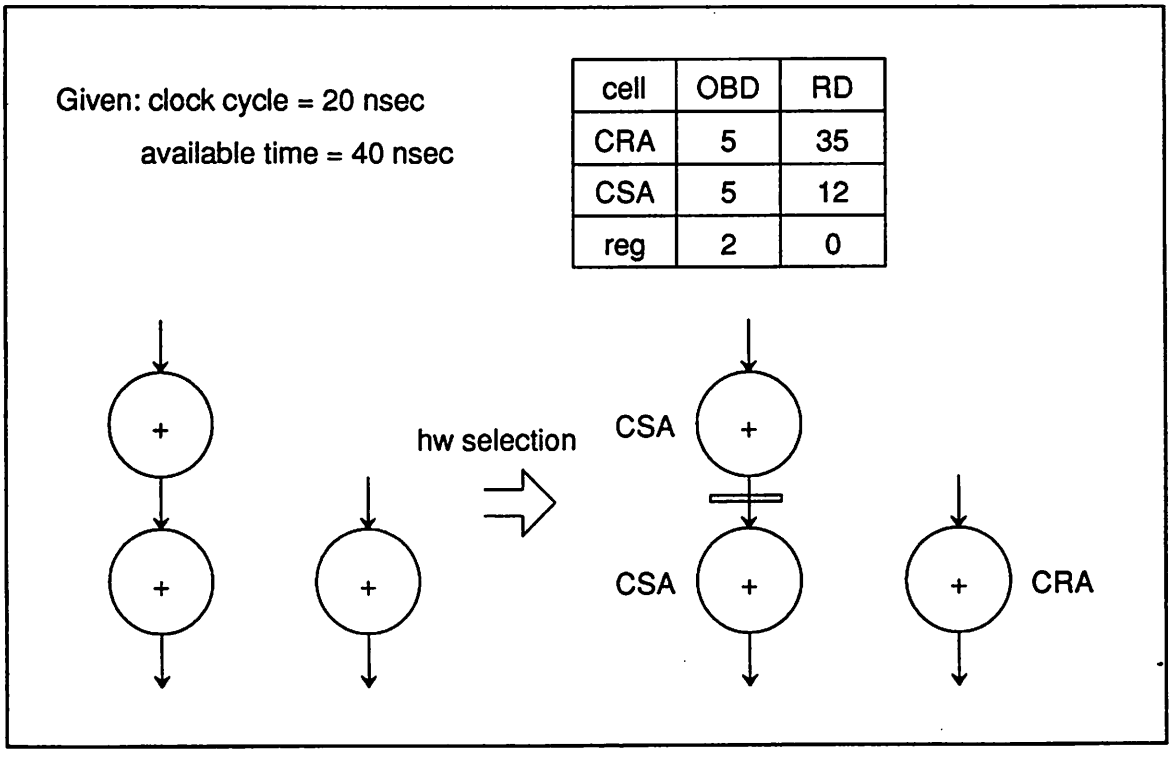

Figure 3.19: Hardware Swapping to Meet Throughput Constraints

Chapter 4

Test Examples and Simulation Results

In this chapter, several real-time applications are presented to demonstrate the effectiveness of the HYPER synthesis system, focusing on the hardware selection and the hardware mapping. Layouts of the examples, which are generated by the Lager IV silicon compilation system, are shown to demonstrate the quality of the transformations performed in the hardware mapper. Functional simulation results from the Thor simulator [32] are also discussed to verify the correctness of the examples. The examples shown in this Chapter range from general-purpose filters, such as the IIR filter and the FIR filter, to application specific processors, such as the Viterbi processor. One of the examples is the standard benchmark from the High Level Synthesis Workshop [16], while the others are carefully designed for real applications.

4.1 Epsilon Processor

Epsilon processor is a part of a real-time, large-vocabulary, continuous speech recognition system [64] [67] based on the ϵ model[1]. The function of the Epsilon processor can be described by the following formula:

$$P_\epsilon = MAX_{i=1}^{N}(P_i \times \epsilon_i)$$

[1]The ϵ model is used to alleviate the excessive computational and storage requirements associated with modeling a fully connected graph in the speech recognition problem.

Where N is the vocabulary size, P_i is the probability that the word i ends at a particular point of time t, ϵ_i is the probability out of the ith word, and P_ϵ is the output probability. After calculating the output probability by the above formula, we can multiply the output probability by ϵ_j (representing the probability into the jth word) to get the probability of the jth word at time $t+1$.

To implement the Epsilon function on a customized processor, several architectural decisions are made:

1. To minimize the amount of hardware, the logarithm of the probabilities is used. The multiplication in the above equation can therefore be reduced to an addition.
2. ϵ_i's are stored in a single off-chip RAM memory.
3. P_i's are obtained from an off-chip FIFO in an asynchronous fashion.
4. The process starts upon the validation of a RESET signal and the iteration stops when ϵ_i equals a certain value (eg. 3000).

The system clock rate is 5MHz and the throughput constraint of the Epsilon processor is to perform a maximum value of 10,000 iterations in a 10 msec frame. The behavior of the Epsilon processor can be described in Silage as given below:

```
#define MAX_NR_OF_WORDS 3000
#define END_FLAG 0xFF
#define bit fix<1, 0>

func main(reset : bit; FWP: fix<32, 0>;
          eps1: fix<14, 0>[MAX_NR_OF_WORDS]) eof_flag: bit =
begin
    while (reset == TRUE) do
    begin
       endflag@@1 = FALSE;
       i@@1 = 0;
       pb_eps1@@1 = 0;
       bt_eps1@@1 = 0;
```

```
    end;

    while (i < MAX_NR_OF_WORDS && endflag == FALSE) do
    begin
       i = i@1 + 1;
       (bt_in, pb_in) = (FWP<31..14>, FWP<13..0>);
       endflag = (eps1[i] == END_FLAG);
       tmp = pb_in + eps1[i];
       (pb_eps1, bt_eps1) = (tmp >= pb_eps1@1 && endflag == FALSE)?
                            (tmp, bt_in) : (pb_eps1@1, bt_eps1@1);
    end;

    eof_flag = endflag;
end;
```

From the Silage description, a flow graph is generated by a one-to-one mapping process with some simple compiler transformations. The flow graph is shown in Figure 4.1. The basic structure of the flow graph can be divided into three parts: The first part is the initialization process of the algorithm, which is represented by the first while node. The second part is the main loop of the algorithm, which iteratively finds the maximal output probability and is represented by the second while node in the figure. To achieve the throughput constraint, the central loop has to be scheduled as compact as possible. The last part of the algorithm is represented by a func node in which postprocessing such as saving the computation result or signaling end-of-process is performed.

One particular implementation of the Epsilon processor is to allocate hardware to each operation such that no resource is shared. This implementation achieves the maximal speed. The data path structure and the state transition diagram of this implementation are shown in Figure 4.2 and Figure 4.3 respectively. This design has been partitioned into five data paths based on the partitioning algorithm. The layout produced by the Lager system is given in Figure 4.4. The area is 2.95 mm^2 assuming the 2-micron CMOS technology.

Another possible implementation of the Epsilon processor shares some hardware resources so that the hardware cost can be reduced. With the same flow graph as shown

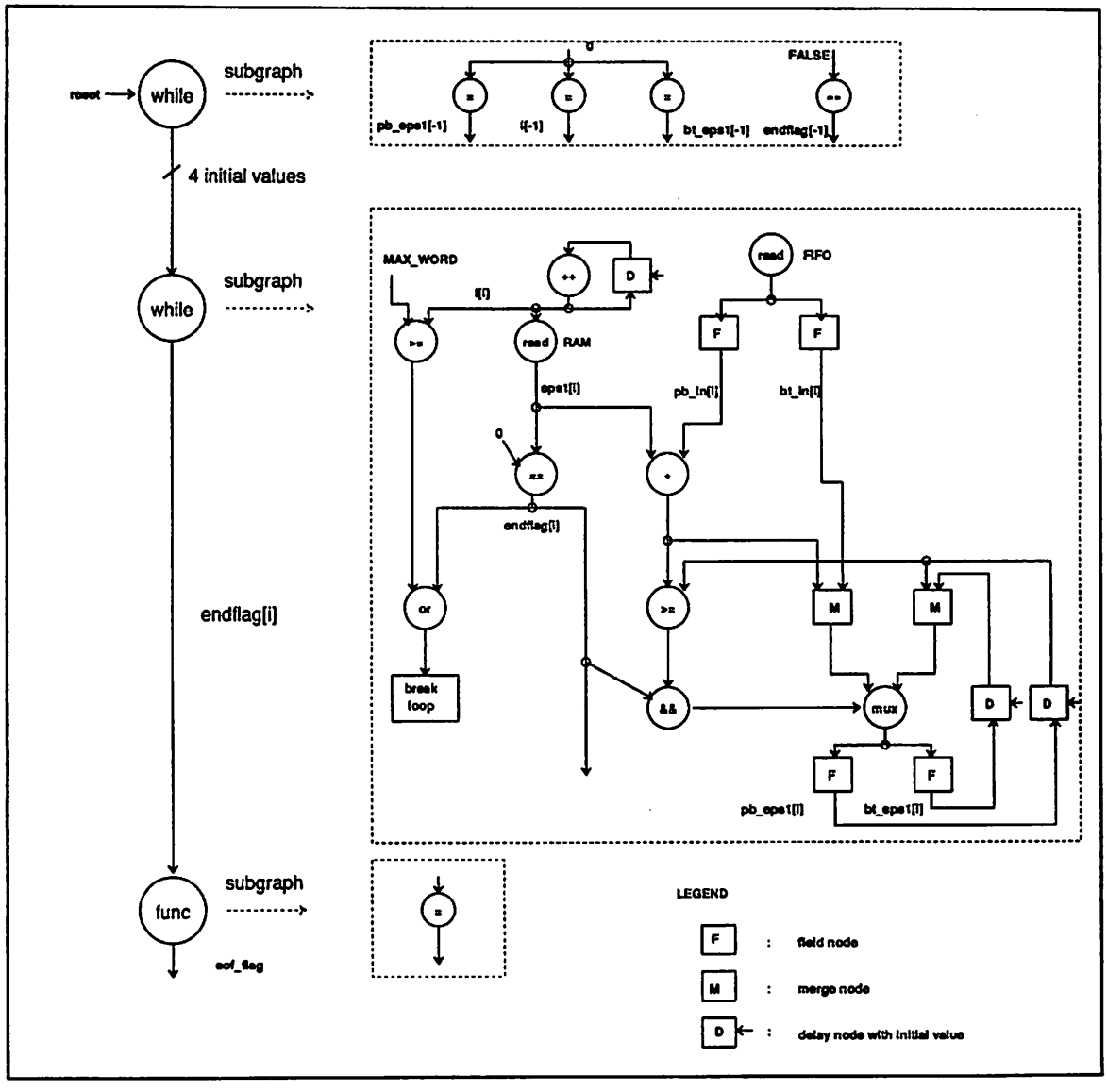

Figure 4.1: CDFG of the Epsilon Processor

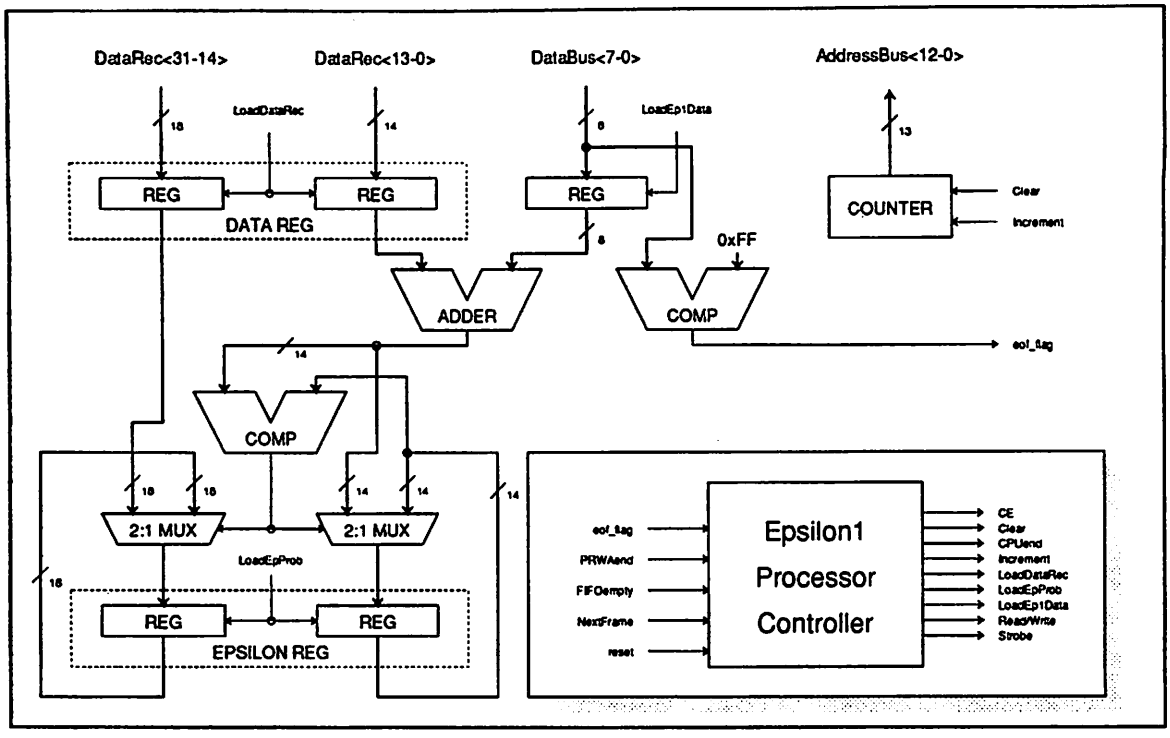

Figure 4.2: Data Path of the Epsilon Processor

in Figure 4.1, we modified the scheduling and allocation and obtained a different design. In this design, the scheduling of the central loop cannot be as compact as the first implementation. Five clock cycles are required for each iteration instead of four clock cycles as obtained by the first implementation. Furthermore, some overhead such as irreducible multiplexers is introduced due to resource sharing. Table 4.1 shows the timing and area tradeoffs of these two implementations.

Comparing with the manual design of the Epsilon processor, the HYPER design has a smaller area even though both designs have the same resource allocation. The area

	description	exu's	area ratio	main loop timing
imp 1	dedicated HW	1 adder, 1 counter 2 comparators	1 ($2.95mm^2$)	4 cycles
imp 2	resource sharing	1 adder, 1 comparator	0.92	5 cycles

Table 4.1: Comparison of Two Epsilon Processor Implementations.

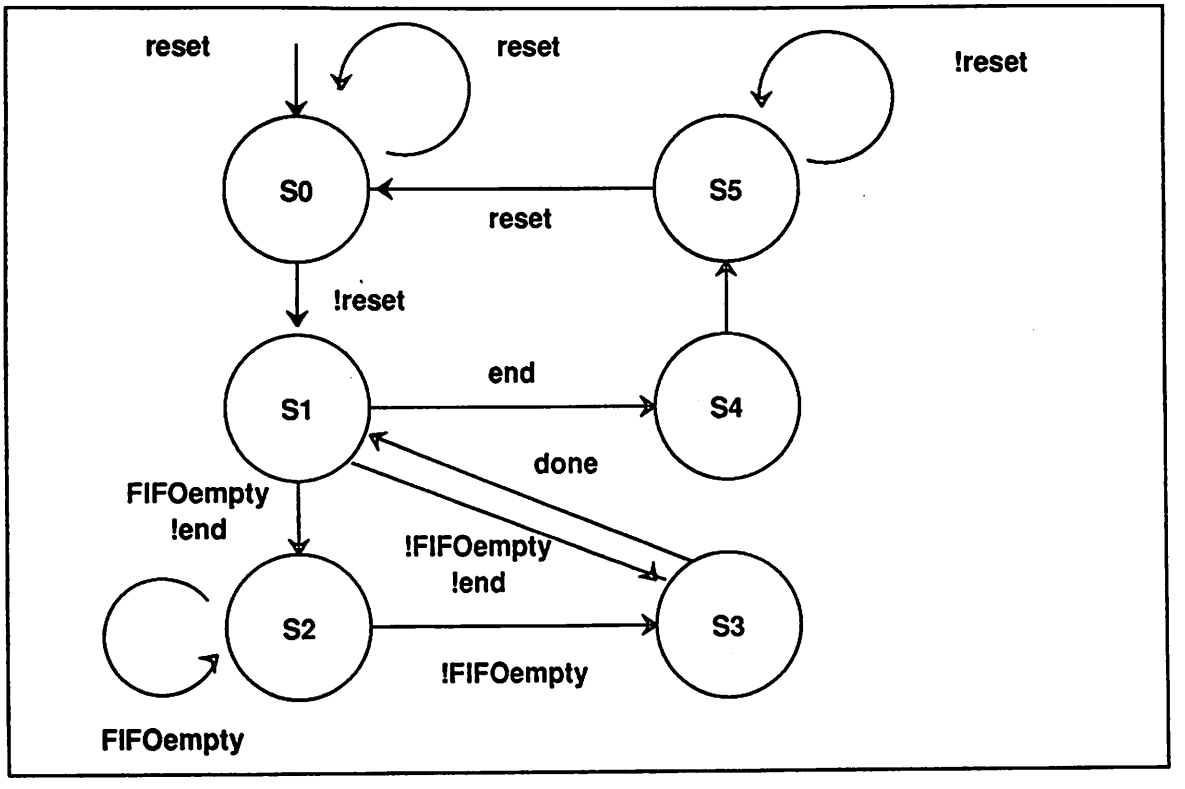

Figure 4.3: State Transition Diagram of the Epsilon Processor

difference is due to the fact that the manual design uses scan registers for testing, while the HYPER design doesn't have the built-in test.

Functional level simulation has been performed on the Epsilon processor to verify the correctness of the synthesis process. Several problems, including the processor initialization and the semantics of the control macros, were detected through the simulation. After the modification of the design, the simulation succeeded and the correctness of the synthesis process is proved.

Figure 4.5 shows a snapshot of the simulation. Scope 0 shows the input signals, while Scope 1 shows the state transition of the FSM and the output signals. In Scope 0, PHI1 and PHI2 are the two-phase non-overlapped clocks, ctl_net_78 is the reset signal, gl_net4 and gl_net3 are the FIFO inputs, and gl_net0 is the RAM data. In Scope 1, FSM_LOCAL shows the state transition, gl_net5 is the RAM address, ctl_net_72 is the read signal to RAM, ctl_net_83 is the end flag, net3 and net1 are the outputs from the multiplexers in the data path. We can see in the simulation that after the reset signal, the processor enters the main loop, in which the processor reads in data from the RAM and the FIFO and processes

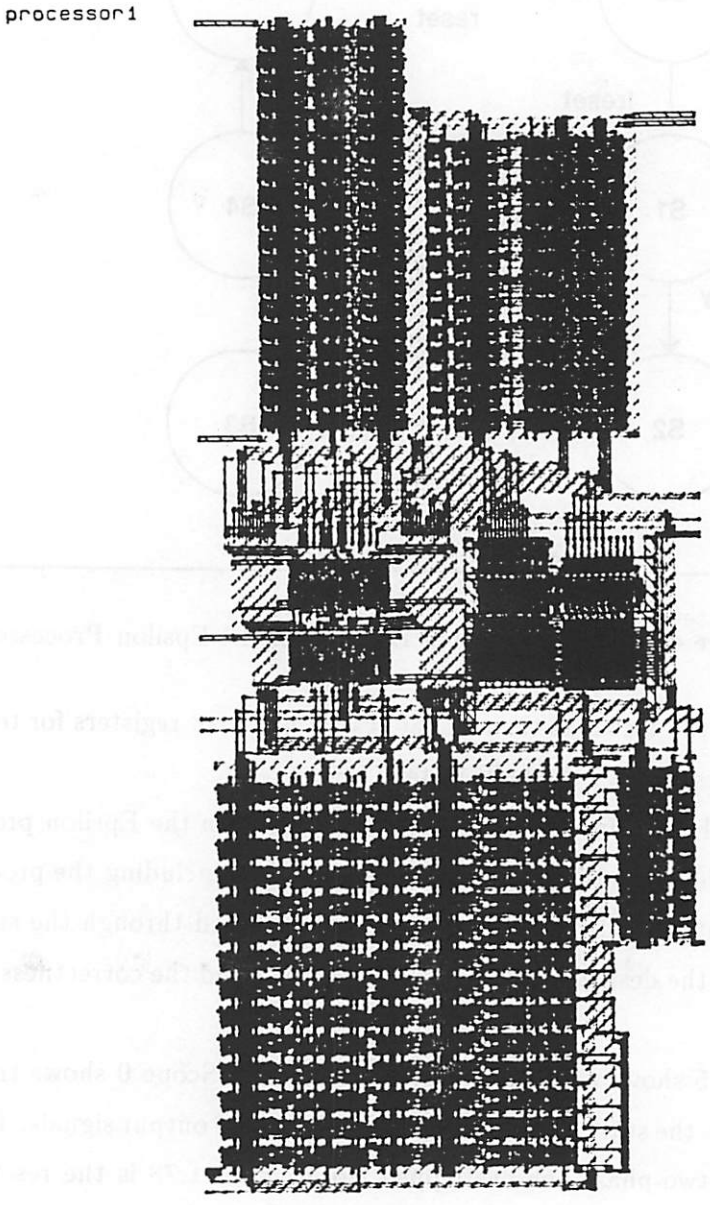

Figure 4.4: Layout of the Epsilon Processor

the data. The main loop terminates when the RAM data is zero and the end flag is set. During the last iteration of the main loop, the multiplexers select the FIFO data and store the data in the Epsilon registers (See Figure 4.2). After one iteration of the algorithm, the processor enters the idle state (state 8) and waits for the next reset signal to be reactivated.

The basic steps to run the simulation are summarized below as a reference [20] [7] [17]:

1. Use DMoct (design manager in the Lager IV system) to generate structure master views (SMV) and structure instance views (SIV) for all the modules of the processor, including data path blocks, FSM, control logic, and memory blocks.
2. Use MakeThorSim to generate simulation models of these modules and the interconnect between them.
3. Attach generators, monitors, and analyzers to appropriate ports. Generators are used to generate input signals; while monitors and analyzers are used to observe the output waveforms.
4. Compile the model files and run the simulation. Sets of test vectors should be carefully designed to cover as many different cases as possible in the simulation.

4.2 Viterbi Processor

Viterbi processor is a special purpose VLSI processor developed for a real-time speech recognition system based on the Hidden Markov Model (HMM) [18]. The function of the Viterbi processor can be described by the following equation:

$$P(O_i, s) = MAX_p[P(O_{i-1}, p) \times A(p, s)] \times P(o_i|s)$$

where s is the current state, p is the predecessor of state s, and o_i is the feature value at frame i. The above equation states that the state probability of the most probable state sequence that ends in s and generates O_i (denoted by $P(O_i, s)$) is equal to the state probability of the previous state p (denoted by $P(O_{i-1}, p)$) times the transition probability between p and s (denoted by $A(p, s)$) times the output probability $P(o_i|s)$.

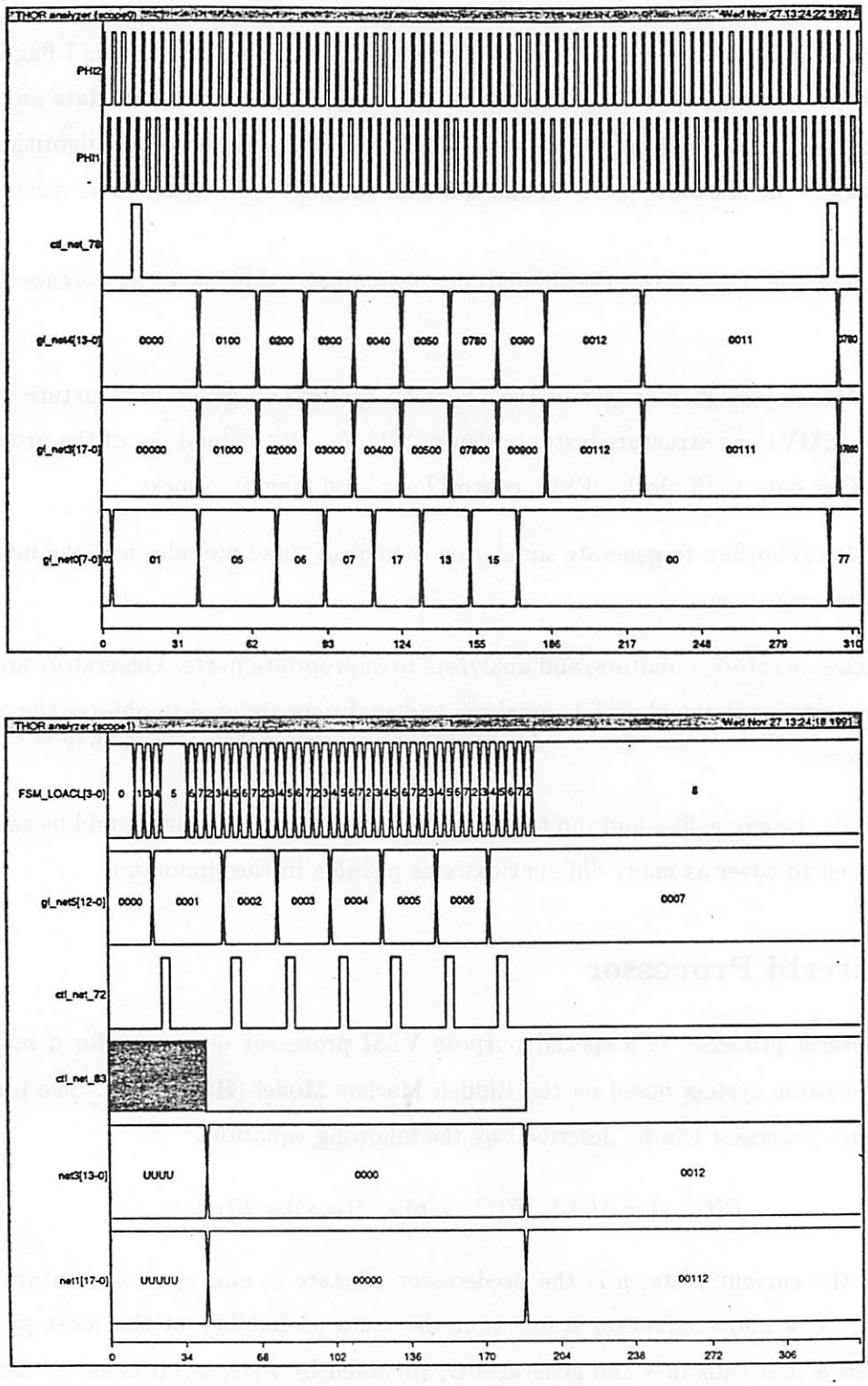

Figure 4.5: Simulation of the Epsilon Processor

The Viterbi algorithm is computationally intensive and requires high throughput processors to meet the real-time constraint. Several architectural decisions have been made on the design so that a single-chip implementation can meet the throughput requirement.

1. As in the Epsilon processor, we use the logarithm of probabilities so that the multiplications in the above equation are reduced to additions. Since negative logarithm (i.e. $P < 1$) is used, the MAX operation becomes a MIN operation.

2. Three on-chip caches are allocated to store the previous-state probability $P(O_{i-1}, p)$ for fast accessing. All the previous-state probabilities are pre-fetched to the caches from off-chip memories. These caches are two-ported memories so that they can be read and written simultaneously.

3. Off-chip RAM's are used to store $A(p, s)$ and $P(o_i|s)$.

4. Parallel computation is necessary to process the high data rate. Three equivalent parallel data paths are therefore allocated in one processor.

Similar to the design process of the Epsilon processor, a Silage description can be written for the Viterbi processor. The differences between the two processors are, first, the operation of the Viterbi processor is much more complicated than the Epsilon processor. Second, in addition to the loop control macros, if control macros are also included in the Silage description of the Viterbi processor. Finally, the Viterbi processor has three on-chip memory blocks. Due the differences between these two processors, we consider the implementation of the Viterbi processor to be valuable.

After writing the Silage description and generating the flow graph, the synthesis process was executed and the architecture was generated. The simplified flow graph of the Viterbi processor is shown in Figure 4.6, which has a similar structure to the Epsilon processor. Figure 4.7 shows the HYPER generated architecture of the Viterbi processor. Due to the high throughput requirement, this processor is fully pipelined and no hardware resource is shared.

The layout of the Viterbi processor is generated by the Lager system and is shown in Figure 4.8. The area of the core is $28.29mm^2$ and there are 50,000 transistors on the chip. From the layout, we can see that three on-chip caches are generated and the data path is partitioned into five data path blocks. Compared with the hand design, the HYPER design is about 10% larger. The area difference is due to the different controller structures,

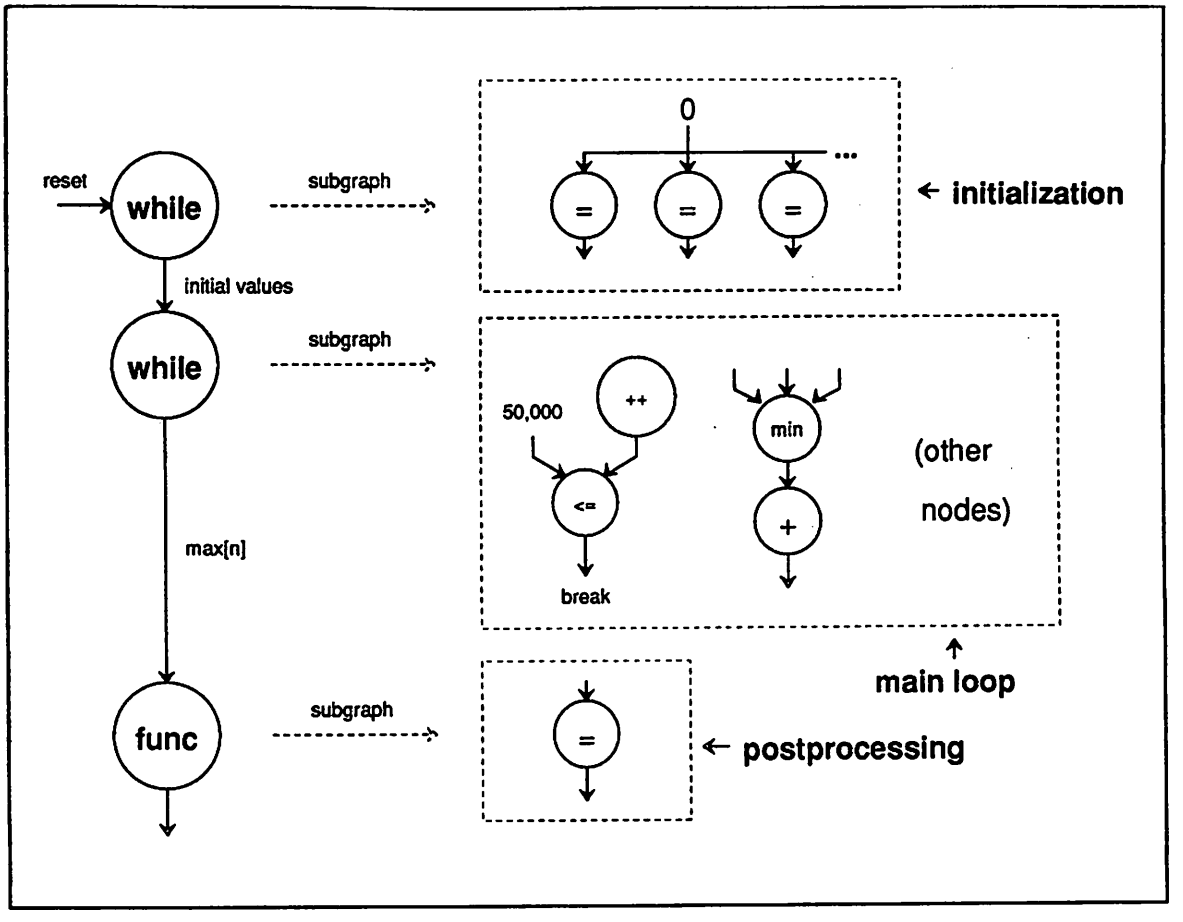

Figure 4.6: Flow Graph of the Viterbi Processor

the different hardware used for certain modules, and the different placement and routing for these two designs.

Similar to the Epsilon processor, functional level simulation has been performed for the Viterbi processor to guarantee the functional correctness of the design. Using the Epsilon processor and the Viterbi processor examples, we successfully demonstrate the correctness and quality of the hardware mapper.

4.3 Infinite Impulse Response (IIR) Filter

A low pass filter with a 6 KHz cut-off frequency, 0.25 dB ripple in the passband, and -80 dB attenuation in the stop band is implemented using HYPER. This filter is a 7th order filter with three biquad's and one first-order filter in series. Each biquad implements

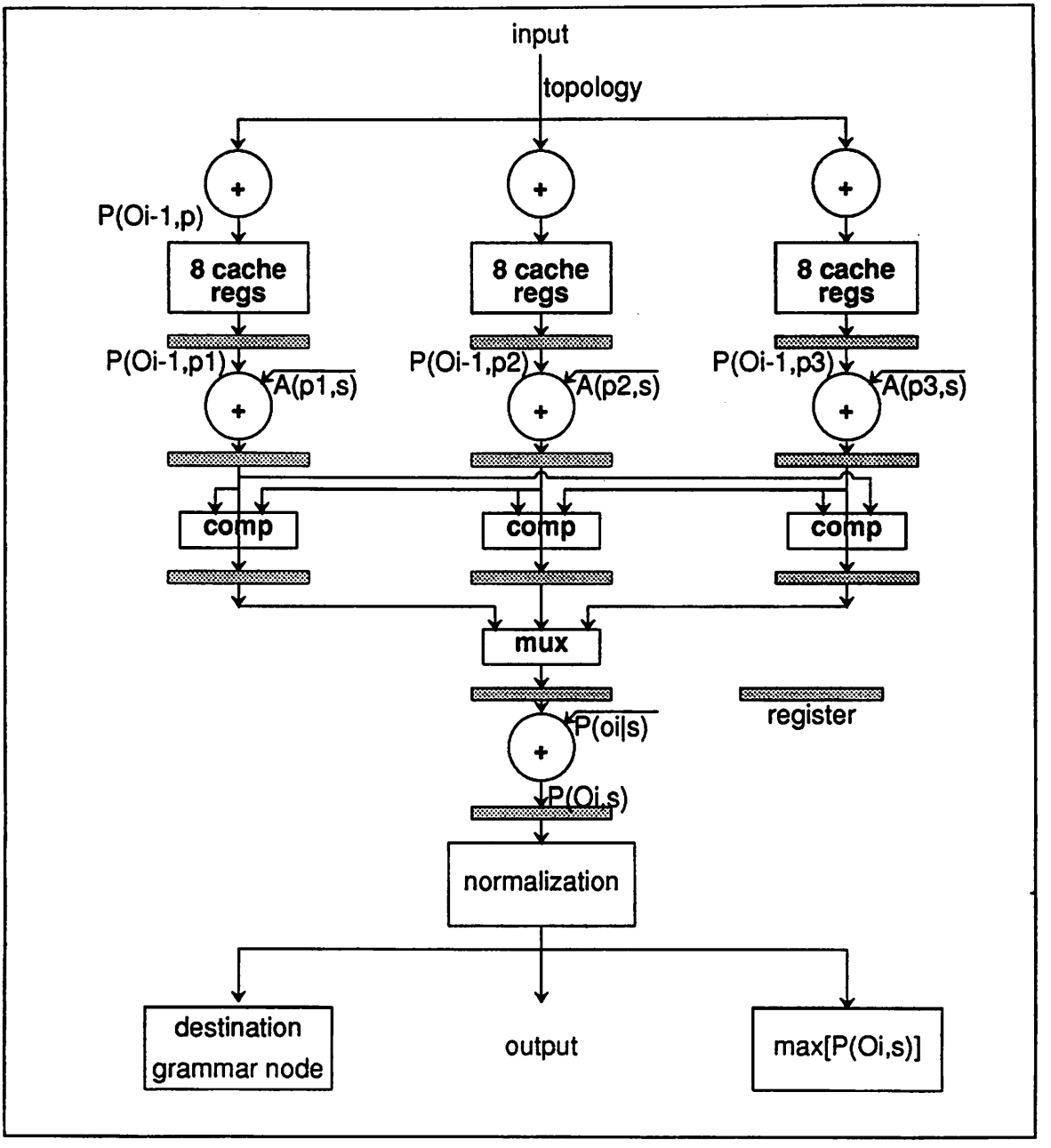

Figure 4.7: Architecture of the Viterbi Processor

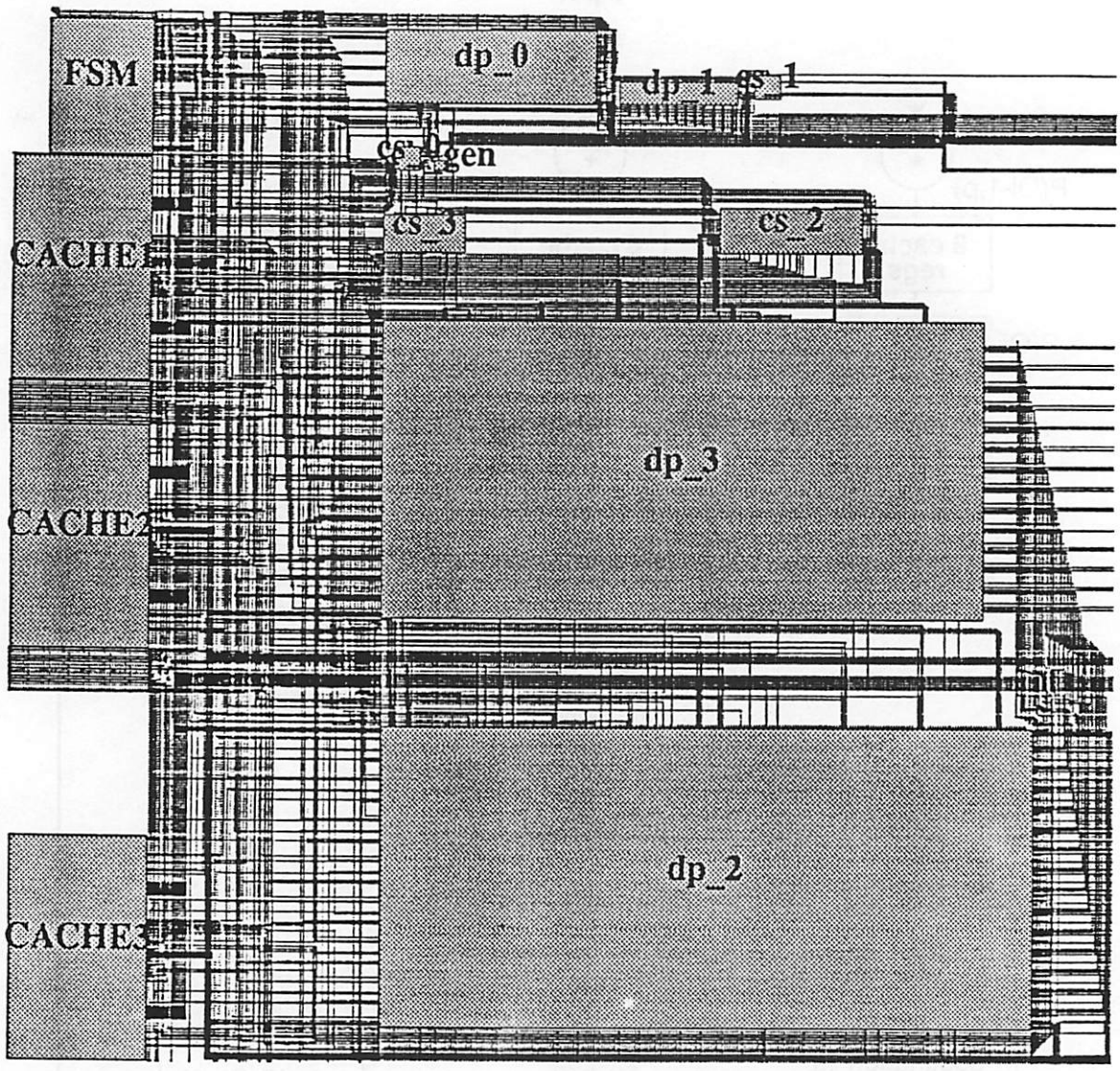

Figure 4.8: Layout of the Viterbi Processor

the following transfer function:

$$\frac{out}{in} = \frac{1 + a_1 z^{-1} + a_2 z^{-2}}{1 + b_1 z^{-1} + b_2 z^{-2}}$$

Where a_i's and b_i's are the coefficients found from the filter synthesis tool FILSYN [1]. The first-order filter is the reduced case of the biquad when a_2 and b_2 are both zero. The coefficients of the first-order filter are also obtained from FILSYN. The Silage description of the IIR filter is described as follows:

```
#define num16 fix<32,10>
#define Coef0 0.001953125
#define Coef1_1 -1.3125
#define Coef1_2 0.625
#define Coef1_3 1
#define Coef1_4 1
#define Coef2_1 -1.25
#define Coef2_2 0.75
#define Coef2_3 0.0625
#define Coef2_4 1
#define Coef3_1 -1.125
#define Coef3_2 0.921875
#define Coef3_3 -0.25
#define Coef3_4 1
#define Coef4_1 -0.71875
#define Coef4_2 1

func main (In : num16) Out : num16 =
begin
    In1 = num16(In*Coef0);
    In2 = biquad(In1, Coef1_1, Coef1_2, Coef1_3, Coef1_4);
    In3 = biquad(In2, Coef2_1, Coef2_2, Coef2_3, Coef2_4);
    In4 = biquad(In3, Coef3_1, Coef3_2, Coef3_3, Coef3_4);
    Out  = firstorder(In4, Coef4_1, Coef4_2);
```

```
end;

func biquad(in, a1, a2, b1, b2 : num16) : num16 =
begin
    state@@1 = 0.0;
    state@@2 = 0.0;
    state = in - (num16(a1*state@1) + num16(a2*state@2));
    return = state + (num16(b1*state@1) + num16(b2*state@2));
end;

func firstorder(in, a1, b1: num16) : num16 =
begin
    state@@1 = 0.0;
    state = in - num16(a1*state@1);
    return = state + num16(b1*state@1);
end;
```

Based on the Silage description, the flow graph of the IIR filter is obtained and is shown in Figure 4.9. In this figure, triangles represent constant multiplications and D's represent delay nodes. These constant multiplications can be expanded to add-shift's in the real implementation. A program called CANDI [35] is used to find the minimal number of add-shift's required for the multiplications.

The Silage description of the IIR filter is first simulated algorithmically to ensure that the design meets the performance specification. After the simulation, hardware selection and transformations such as multiplication expansion and retiming are performed. Estimation is done each time when a new flow graph structure is generated after the transformation. When the estimated quality is satisfied, allocation and scheduling will be performed and a decorated flow graph is generated. Hardware mapper then takes the decorated flow graph and produces all the files required for layout generation. For the IIR example, four implementations with different allocation and scheduling were generated using HYPER. Table 4.2 shows the timing and area tradeoff of the four implementations. Three layouts of the four implementations are shown in Figure 4.10. These layouts are properly scaled to show their relative sizes. The area grows almost linearly when the computation time goes

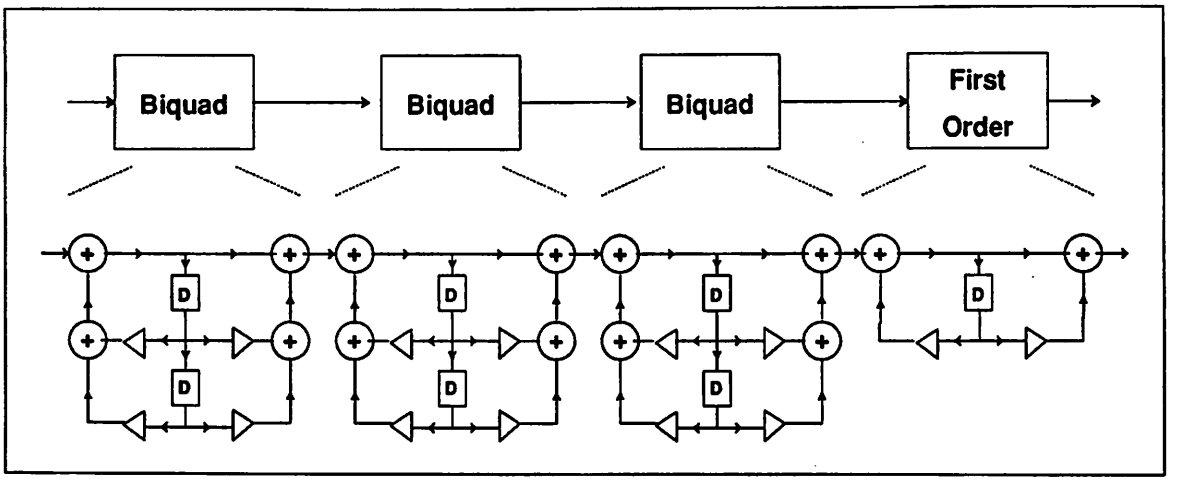

Figure 4.9: Flow Graph of the 7th Order IIR Filter

	imp 1	imp 2	imp 3	imp 4
clock cycles	20	16	13	10
adder	1	2	2	2
subtracter	1	1	1	2
barrel shifter	1	1	1	2
register	36	37	41	46
tristate buffer	15	17	16	25
area (mm^2)	13	18.9	18.6	27.95

Table 4.2: Comparison of Four IIR Filter Implementations.

down. Table 4.3 summarizes the distribution of the CPU time over the synthesis modules for this particular example on a SPARC II workstation. Table 4.4 summarizes the running time of the major transformations in the hardware mapper.

Another implementation of the same IIR algorithm without the constant multiplication transformation is also synthesized using HYPER. Two multipliers are needed for this implementation and the throughput is 13 clock cycles per sample. Figure 4.11 shows the layout of this implementation. In addition to the two array multipliers, one adder, one subtracter, 31 registers, and 11 tristate buffers are allocated. The chip area is $32mm^2$ [2] and about 70% larger than the implementation with the constant multiplication expansion.

[2]This is a 16-bit implementation.

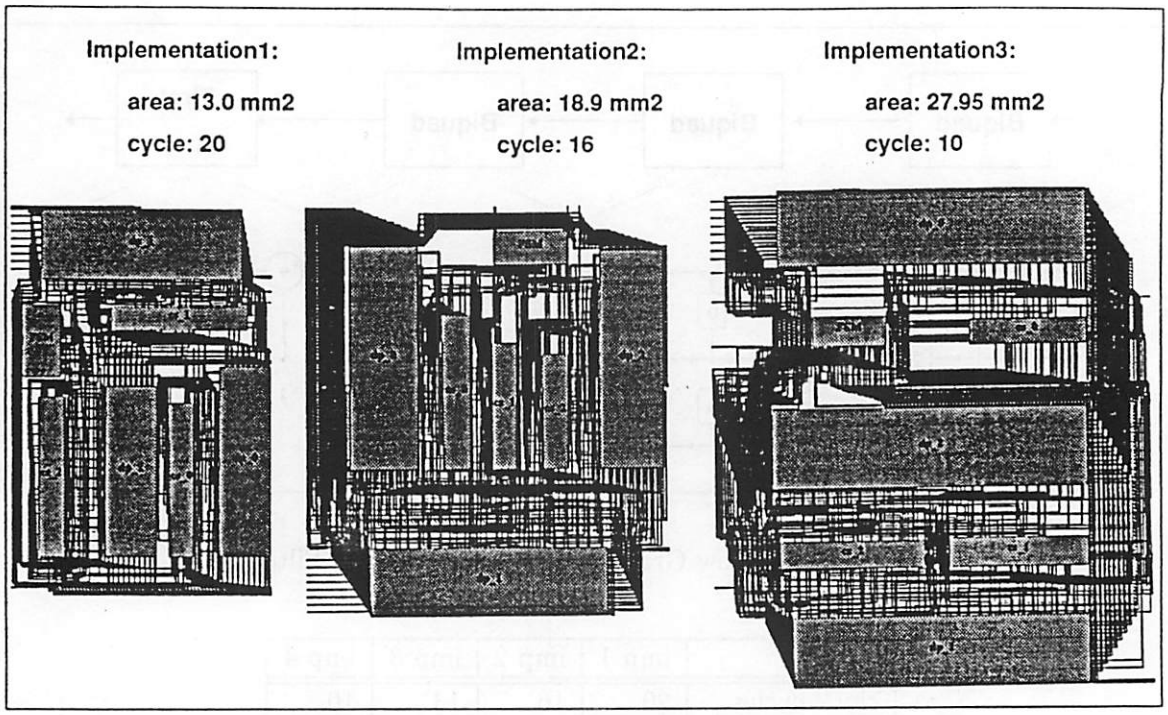

Figure 4.10: 3 IIR Filter Layouts of Different Implementations

The benefit of the constant multiplication expansion is successfully demonstrated using this example.

The functional correctness of all produced layouts has been analyzed using the Thor simulator and the simulation results have been checked against the simulation results at the Silage level. This functional simulation has assured us of the correctness of the applied transformations and synthesis operations. Figure 4.12 is the functional simulation result of the IIR filter, which shows the impulse response of the low pass filter. Figure 4.13 is a snap shot of the simulation. In this figure, PHI1 and PHI2 are the two-phase non-overlapped clocks, FSM_LOCAL is the state transition of the FSM, gl-net0 is the input signal (an impulse), and gl-net1 is the output of the filter.

The IIR example has demonstrated that designers can easily compare the tradeoffs of various implementations using the HYPER synthesis system. Optimizations at the flow graph level do not necessarily reduce the chip area, nor does less hardware allocation. Layout level issues such as data path partitioning, placement and routing play a very important role. How to predict the chip area at the flow graph level without generating real layouts is

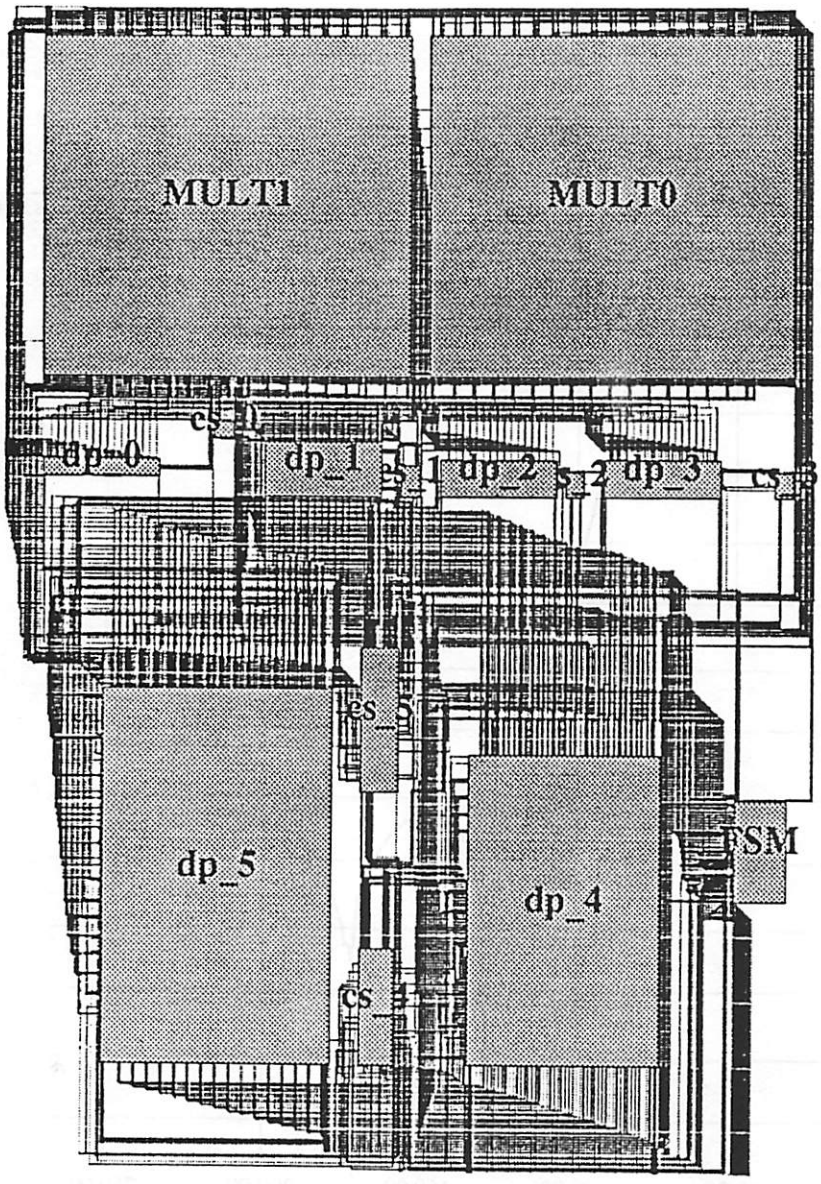

Figure 4.11: Layout of the IIR Filter with Multipliers

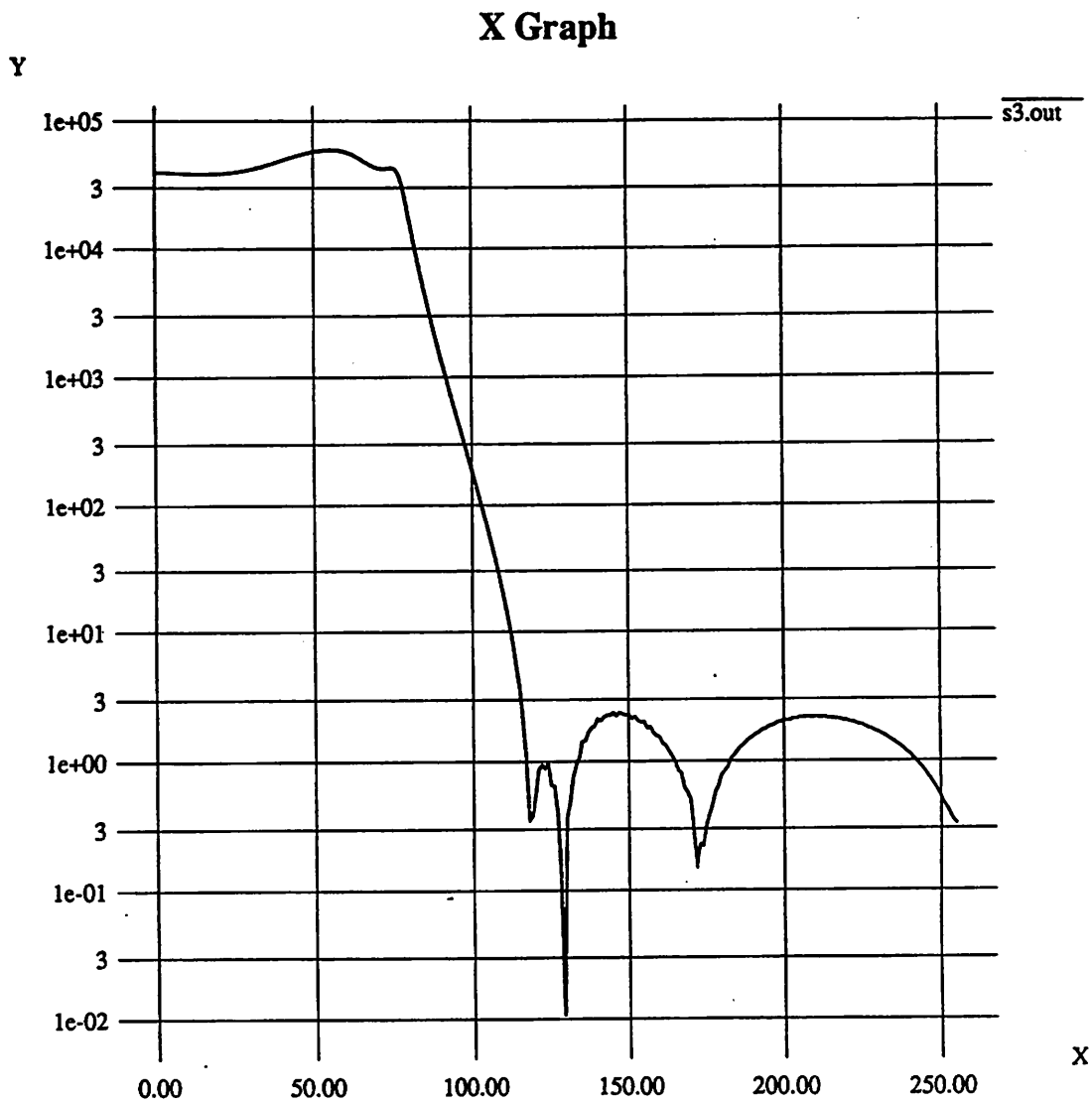

Figure 4.12: Impulse Response of the IIR Filter

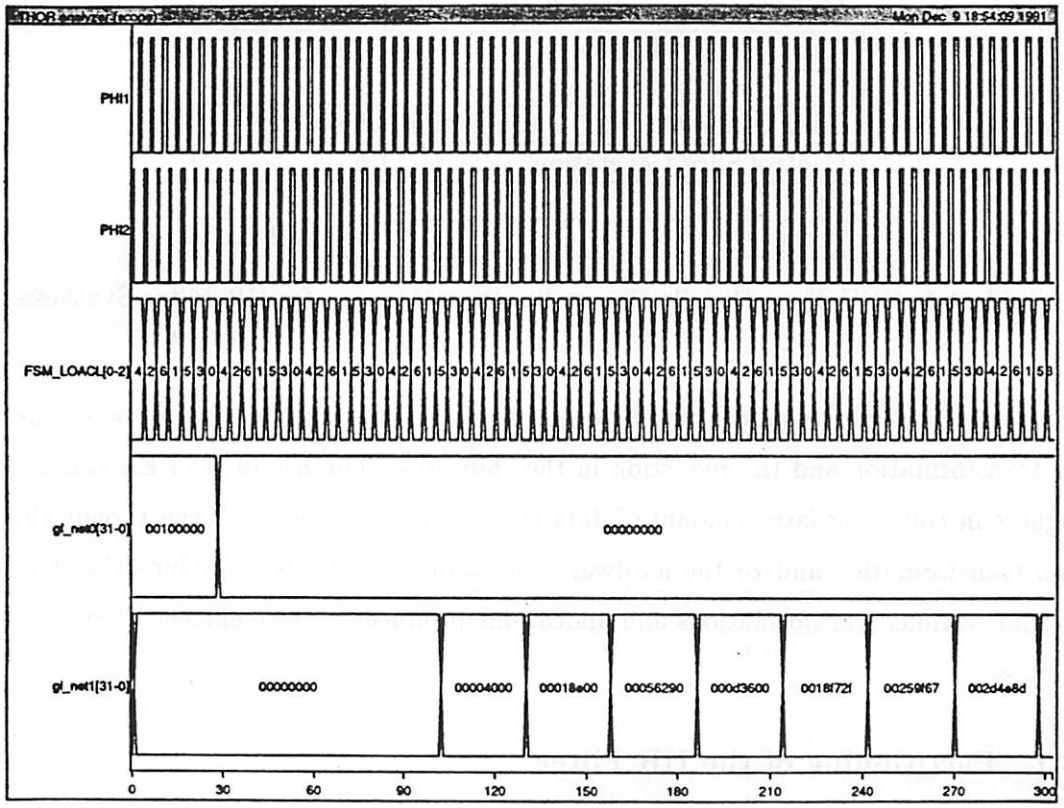

Figure 4.13: Snap Shot of the IIR Filter Simulation

Flow Graph Generation	0.8 sec
Module Selection	1.8 sec
Estimation/Allocation	1.7 sec
Retiming	7.4 sec
Assignment/Scheduling	1.9 sec
Hardware Mapping	about 2 min
Layout Generation	about 1 hour

Table 4.3: CPU Time Distribution for IIR Filter Synthesis (on SUN 4/100)

Lisp Set-up and Compilation	about 1 min
Register File Merging	7 sec
Data Path Partitioning	17 sec
FSM Generation and Optimization	8 sec
Control Slice Generation	5 sec
Other Tasks	10 sec
Total	about 2 min

Table 4.4: CPU Time Distribution in Hardware Mapper for IIR Filter Synthesis

difficult. A probabilistic model may be useful to analyze the relationship between the high level transformation and the reduction in the chip area. The use of HYPER can help the designers in collecting large amount of data to perform the analysis. Even though the flow graph transformation and/or the hardware allocation do not always reduce the chip area, exploring various transformations and allocations produces a more efficient design most of the time.

4.3.1 Partitioning of the IIR Filter

In addition to comparing the tradeoffs of different resource allocation, HYPER also helps in exploring various layout alternatives. For example, we can specify the number of data paths in the hardware mapping process to improve the layout efficiency. Taking the second implementation from Table 4.2, three possible designs with different data path partitioning are generated. Table 4.5 lists the number of data nets, the number of control nets, and the areas of these three designs. Figure 4.14 shows the layouts of these IIR filters,

# Data Paths	# Data Buses	# Control Nets	Area (mm^2)	Area ratio
two	12	264	22.11	1.17
three	15	266	18.9	1
four	12	269	23.21	1.228

Table 4.5: Partition of the IIR Filter

which are properly scaled to show their relative sizes[3]. For this example, the three-datapath design is the most efficient one. This design is also the result from the automatic data path partitioning.

Although layout efficiency depends heavily on the performance of the placement-and-routing tool, data path partitioning also plays an extremely important role. A good data path partitioning not only reduces the number of nets, but also produces proper sizes of data paths. Since the performance of data path partitioning can only be verified by the final layouts, the combination of the HYPER tool and the Lager tool provides a useful mechanism to explore the design space. In conclusion, the experiment described in this section is interesting because it demonstrates the capability of exploring the design space by the synthesis tools. Furthermore, it also verifies the quality of the data path partitioning process in the hardware mapper.

4.4 Finite Impulse Response (FIR) Filter

This section describes the synthesis of a 10th order FIR filter, which performs the $x/sin(x)$ function. In general, the transfer function of a FIR filter can be described by the following formula:

$$\frac{out}{in} = \sum_{i=1}^{10} c_i z^{-i}$$

Where c_i's are the constant coefficients found from FILSYN. The Silage description of the filter is shown below and the flow graph is given in 4.15. This FIR filter example is very similar to the IIR filter example as described above in the synthesis process. The purpose of studying this example, however, is to test the hardware selection and clustering process of the HYPER system.

[3]These are 8-bit implementations.

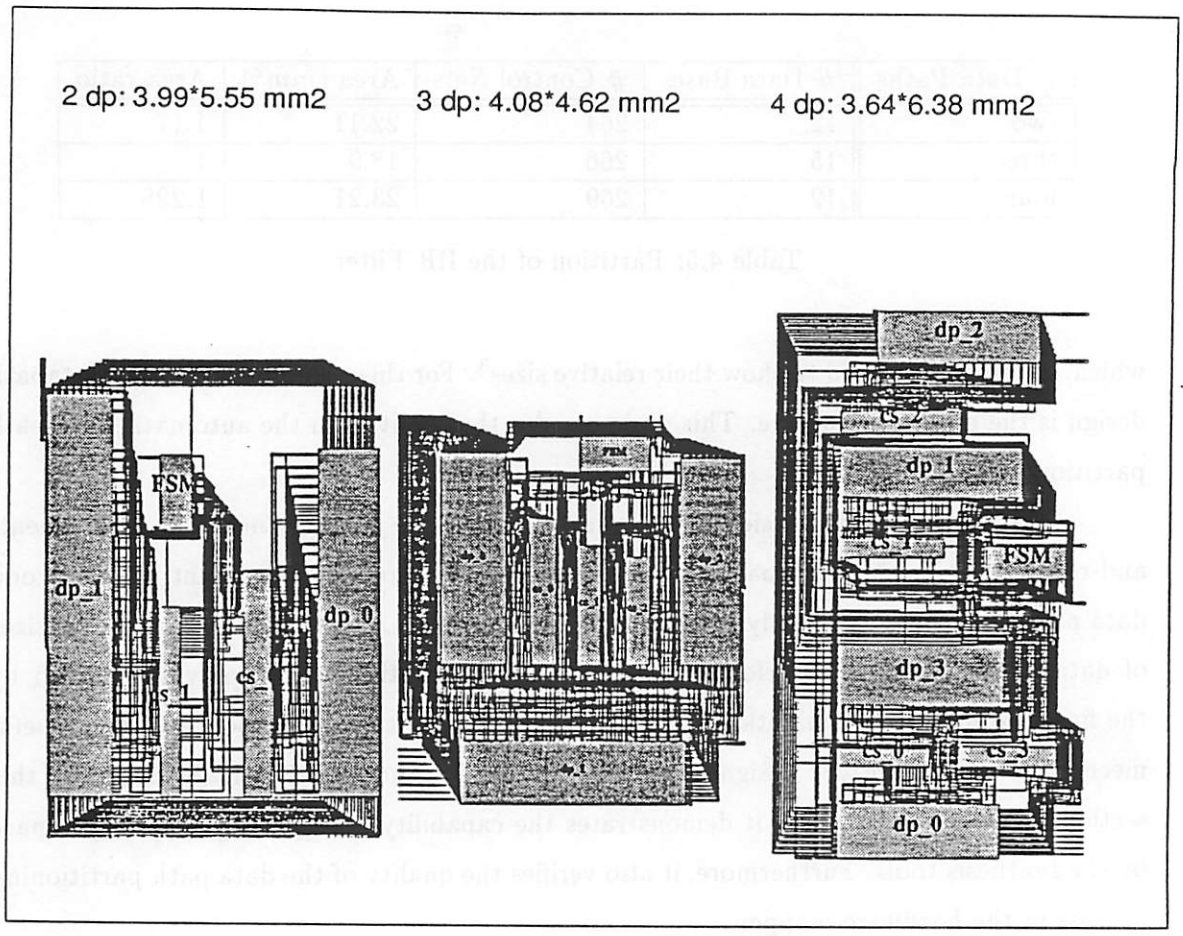

Figure 4.14: Three IIR Filter Implementations to Demonstrate Data Path Partitioning

```
#define num16 num<16,0>

#define a0 -0.001953125
#define a1 0.003906250
#define a2 -0.007812500
#define a3 0.01953125
#define a4 -0.06640625
#define a5 0.7500000

func main(In : num16) Out : num16 =
```

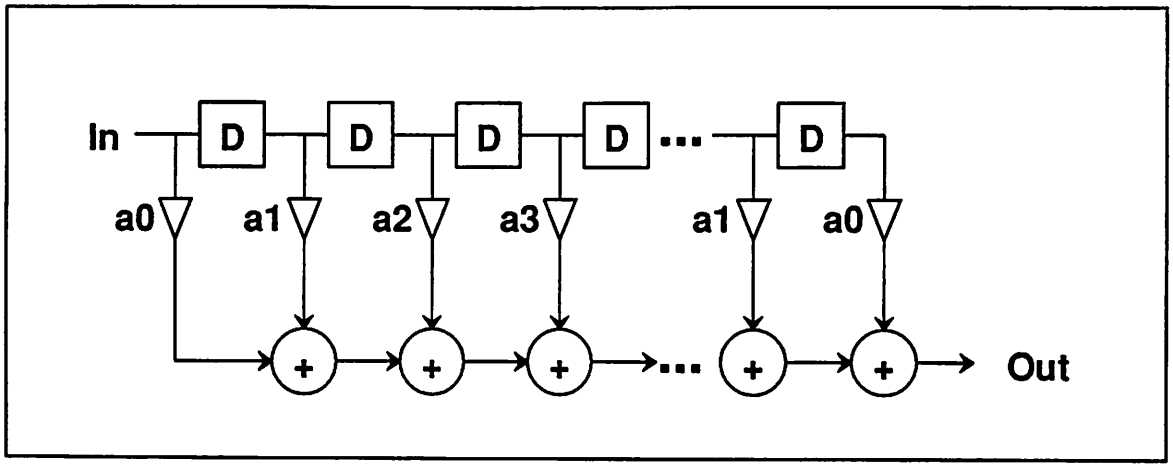

Figure 4.15: Flow Graph of the FIR Filter

```
begin
    Acc1 = num16(In@10 * a0);
    Acc2 = num16(In@9 * a1) + Acc1;
    Acc3 = num16(In@8 * a2) + Acc2;
    Acc4 = num16(In@7 * a3) + Acc3;
    Acc5 = num16(In@6 * a4) + Acc4;
    Acc6 = num16(In@5 * a5) + Acc5;
    Acc7 = num16(In@4 * a4) + Acc6;
    Acc8 = num16(In@3 * a3) + Acc7;
    Acc9 = num16(In@2 * a2) + Acc8;
    Acc10 = num16(In@1 * a1) + Acc9;
    Out = num16(In * a0) + Acc10;
end;
```

The critical path of the flow graph is 11 clock cycles. After clustering, the critical path reduces to 9 clock cycles. Table 4.6 shows the comparison between the two implementations. For this particular example, the clustering process is useful in reducing both the critical path and the hardware cost. The hardware cost after clustering is decreased due to a smaller number of registers required. Figure 4.16 shows the two layouts of the FIR filter. Layout 1 doesn't use the clustering approach, while Layout 2 uses the proposed

	available time	critical path	adder	subtracter	shifter	[>> −]	register
no clustering	15	11	1	1	2	0	33
with clustering	15	9	1	0	1	1	29

Table 4.6: Comparison of Two FIR Filter Implementations.

hardware selection and clustering algorithm. These two layouts are properly scaled to show their relative sizes, which are $24.48mm^2$ and $22.41mm^2$ respectively assuming the 2-micron CMOS technology and 16-bit implementations.

Similar to the IIR example, the functional correctness of the FIR filter is also checked by running the Thor simulation. The impulse response of the FIR filter is shown in Figure 4.17.

4.5 CORDIC Algorithm

CORDIC algorithm [6] [73] [75] is an iterative approach to calculate Polar coordinates from Cartesian coordinates and vice versa. The Silage description and the flow graph of the CORDIC algorithm has been shown in Figure 1.2 and Figure 2.1 respectively. As described in Chapter 2, the flow graph includes three major steps represented by three hierarchical nodes – the first *func* node represents the initialization step, the *iteration* node represents the main loop of the algorithm, and the second *func* node represents the post-processing step. Many DSP algorithms have a similar flow graph structure as the CORDIC flow graph and therefore they can be handled in a similar fashion. The goal of experimenting the CORDIC algorithm is to test the capability of HYPER in handling hierarchical flow graphs.

The non-pipelined design of the algorithm has a critical path of 88 clock cycles. Given a throughput constraint, say 150 clock cycles per sample, the estimator can distribute the available time over different hierarchical nodes according to a heuristic measure, called the *criticality*[4] of the nodes. Then the scheduler/allocator can perform hierarchical scheduling from the estimation result. The scheduler is able to complete the scheduling with the lower bound of the hardware. However, the resource utilization is low due to the fact that

[4]The criticality of a hierarchical node is a function of the critical path and the number of nodes of the subgraph of the hierarchical node. It is similar to the stress discussed in Chapter 3.

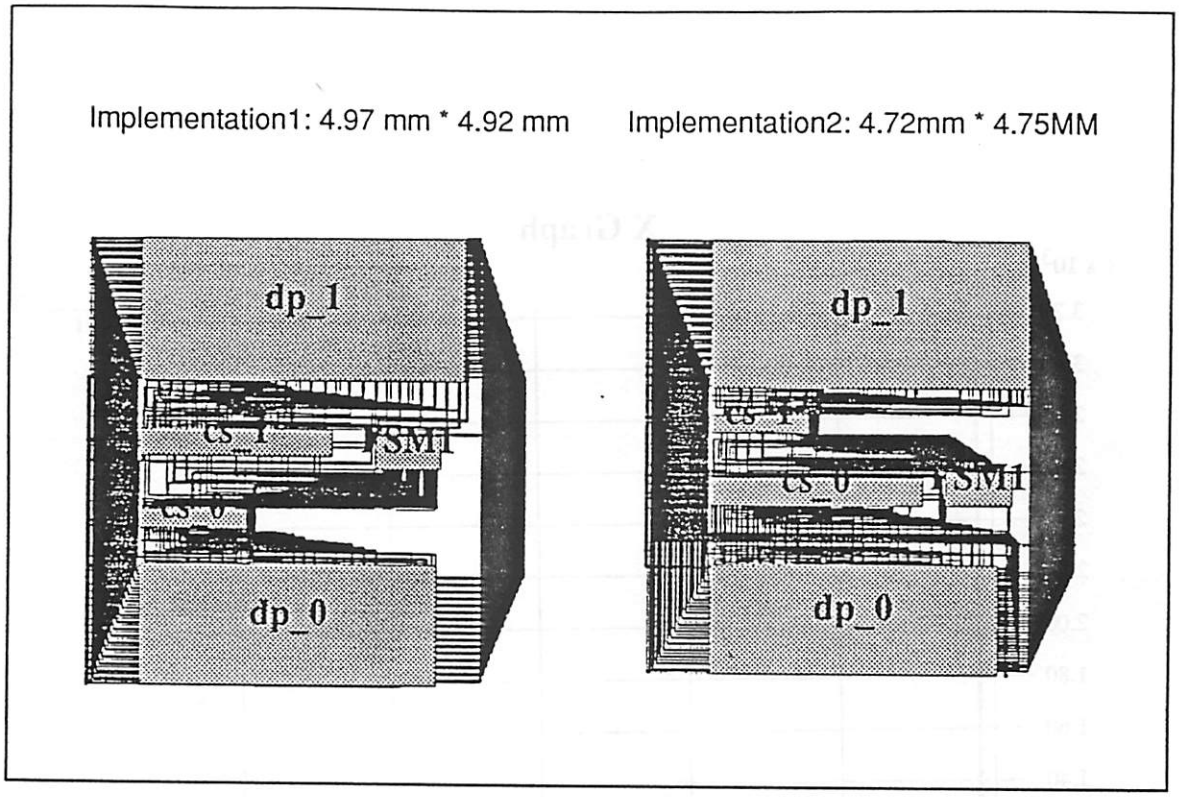

Figure 4.16: Layouts of the FIR Filter

not many execution units can be shared by different operations. Multiple function units should be able to cure this problem. Another Silage description of the CORDIC algorithm with no hierarchical nodes is also written and synthesized using HYPER. The run time for the scheduling and allocation is about 2 minutes, which is much longer than the run time of the hierarchical version (less than 10 sec). But the run time is still fast enough and acceptable for this example.

The synthesis process allocates one subtracter, one adder, one barrel shifter, one comparator, one multiplexer, and one memory unit for the CORDIC processor. After the hardware mapping, the layout of the CORDIC example is generated and as shown in Figure 4.18. The core area is $33mm^2$ assuming a 22-bit implementation and the 2-micron CMOS technology.

The simulation result is shown in Figure 4.19. In this simulation, 8 sets[5] of inputs are tested and the algorithm correctly finds the corresponding amplitude and phase

[5]Each set is a Cartesian coordinate (x, y).

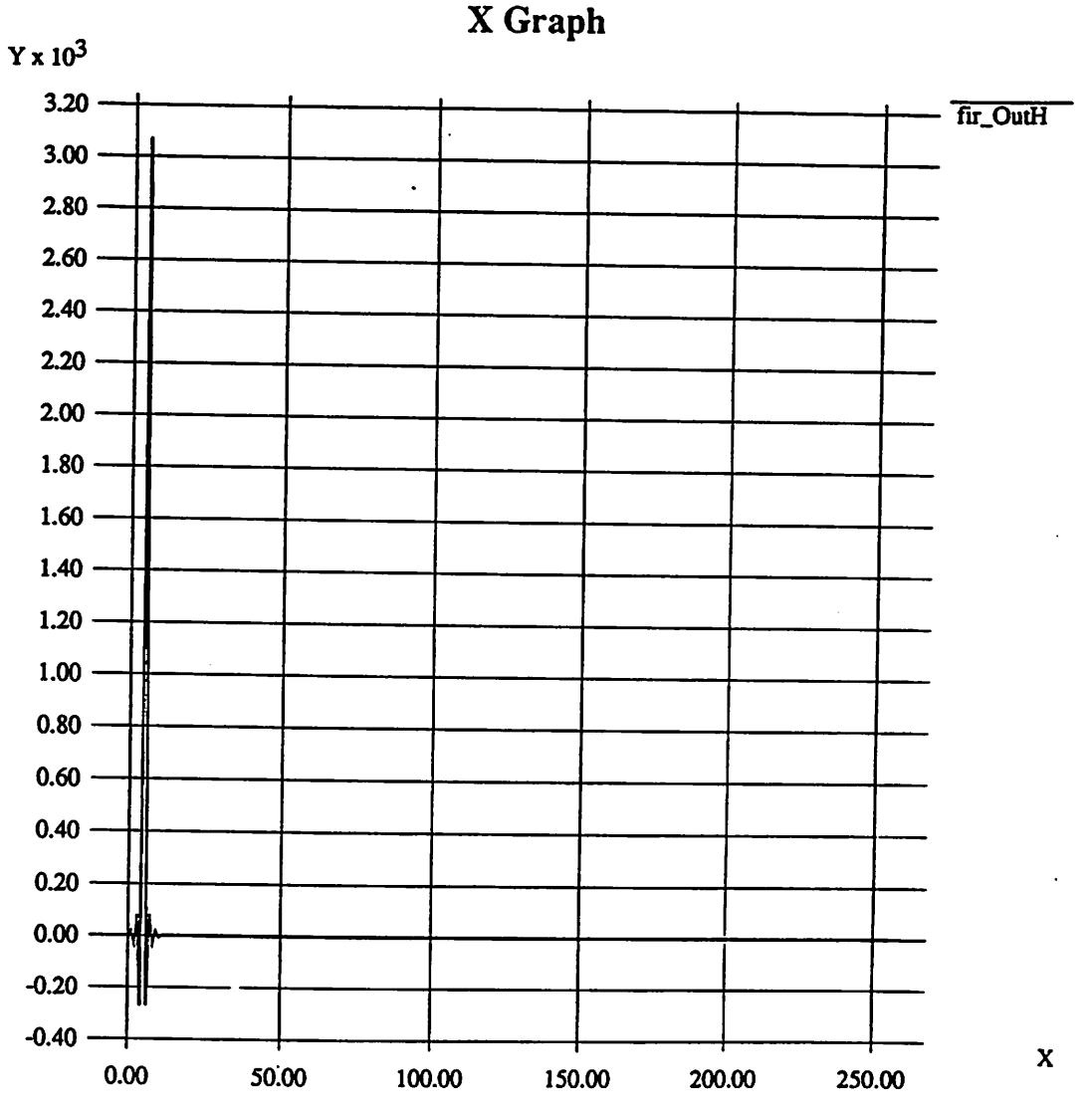

Figure 4.17: Impulse Response of the FIR Filter

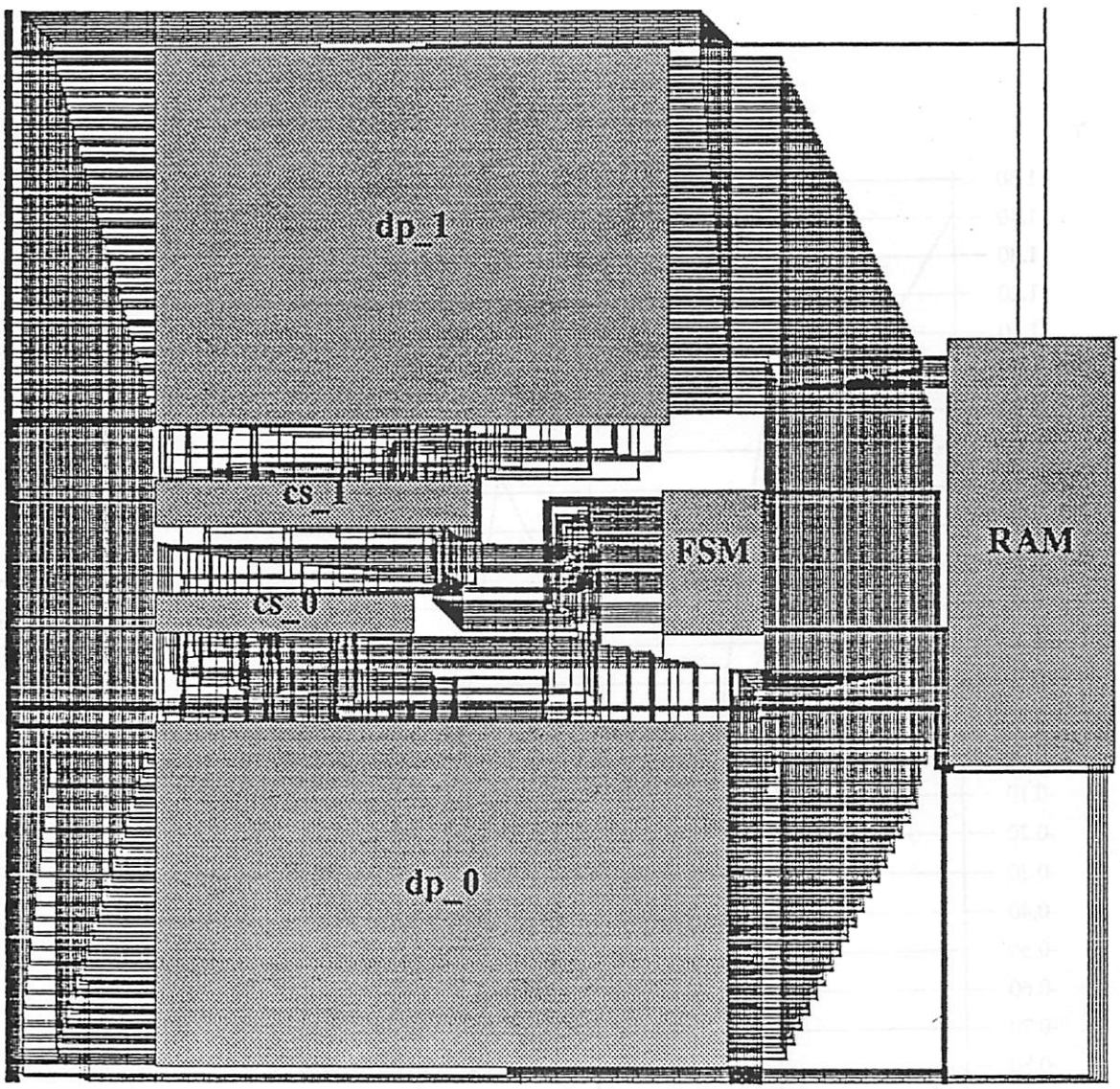

Figure 4.18: Layout of the CORDIC Example

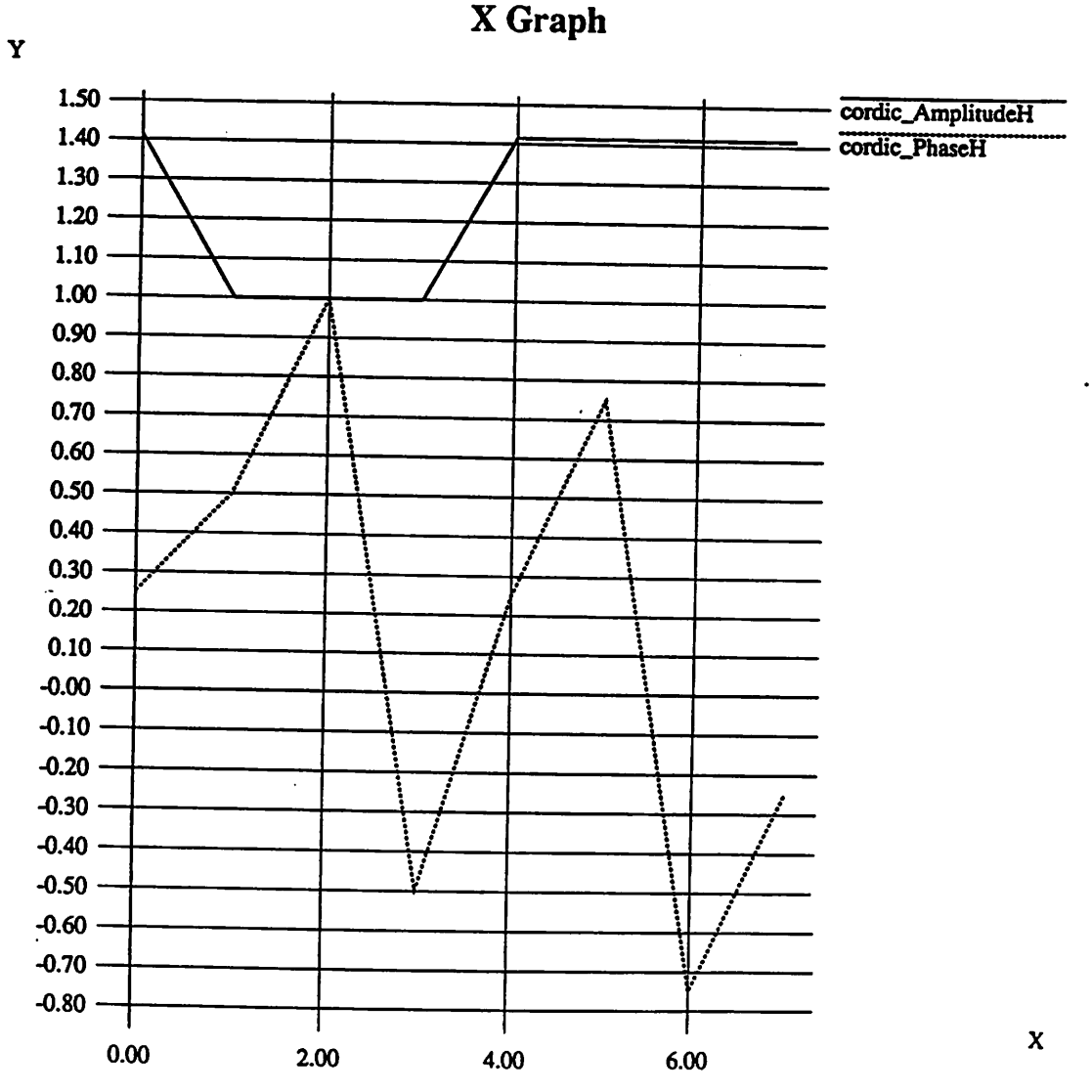

Figure 4.19: Simulation of the CORDIC Example

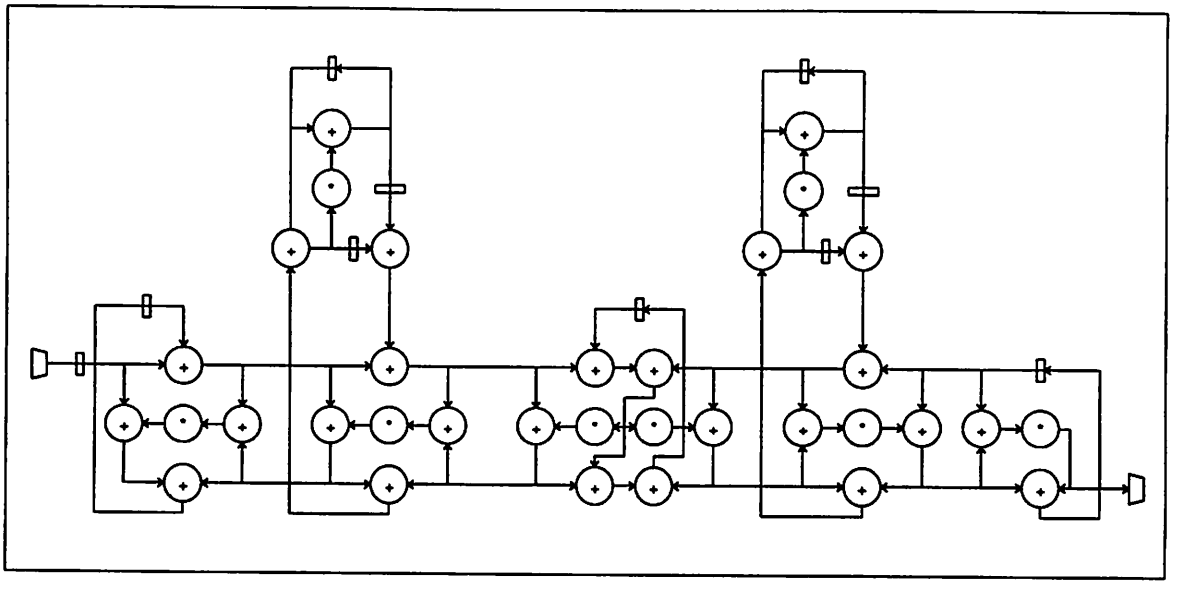

Figure 4.20: Flow Graph of the 5th order WDF

as shown in Figure 4.19, in which the solid line is the amplitude output and the dotted line is the phase output (in π).

4.6 Wave Digital Filter

Wave Digital Filters (WDF) are a class of digital filters with excellent properties. They can be designed using explicit formulas without the knowledge of classical network theory. A good review of the WDF design can be found in [54]. Similar to the FIR filter and the IIR filter design, the coefficients of a WDF can be obtained by running a filter synthesis tool called FALCON [46].

A fifth order elliptic wave digital filter has been chosen as a standard benchmark for high level synthesis [16]. The flow graph of this filter is shown in Figure 4.20. This example has 26 addition nodes and 8 multiplication nodes. Assuming that each addition takes one clock cycle and that each multiplication takes two clock cycles, this filter has a critical path of 17 clock cycles. Since this benchmark is mainly used as a standard for comparing the performance of various scheduling approaches, many design issues such as I/O and the hardware database are not considered.

Unlike [16], the purpose of addressing the wave digital filter in this chapter is to

demonstrate the capability of HYPER in synthesizing this filter. Therefore, we consider many physical design issues including the I/O ports and the cell library. The critical path based on the physical design is 15 clock cycles, rather than 17 clock cycles. Given the sampling period equal to the critical path, 3 adders, 2 multipliers, and 27 registers are allocated without using the proposed clustering approach as described in Chapter 3. With the clustering approach, the hardware required is 2 adders, 1 multiplier, 1 [+ +] unit, and 27 registers. The hardware cost reduces significantly due to the fact that only one multiplier is required. A similar scenario has been discussed in Chapter 3 to demonstrate the hardware cost reduction by clustering.

Given a throughput constraint of 18 clock cycles per sample, two adders, one multiplier, and 23 registers are allocated. The layout can be generated using the hardware mapper and the Lager system as the previous examples and is shown in Figure 4.21. Assuming a 20-bit implementation, the core area is $41.74mm^2$ under the 2-micron technology.

4.7 Conclusion on Test Examples

Several real-time applications are discussed in this chapter. Layouts as well as simulation results of these applications are shown to demonstrate the performance of the HYPER tools, focusing on the hardware mapper and the hardware selector. These examples have successfully demonstrated the quality and correctness of the HYPER design.

Throughout the presentation, we also found many interesting points such as the comparison of the automatic design and the manual design, the almost linearity of the time-area tradeoff of the IIR filters, and the usefulness of the transformations. Many more examples will be designed in the future to further test the performance of HYPER. These tests are essential to improve the quality of the HYPER design and to discover other design alternatives.

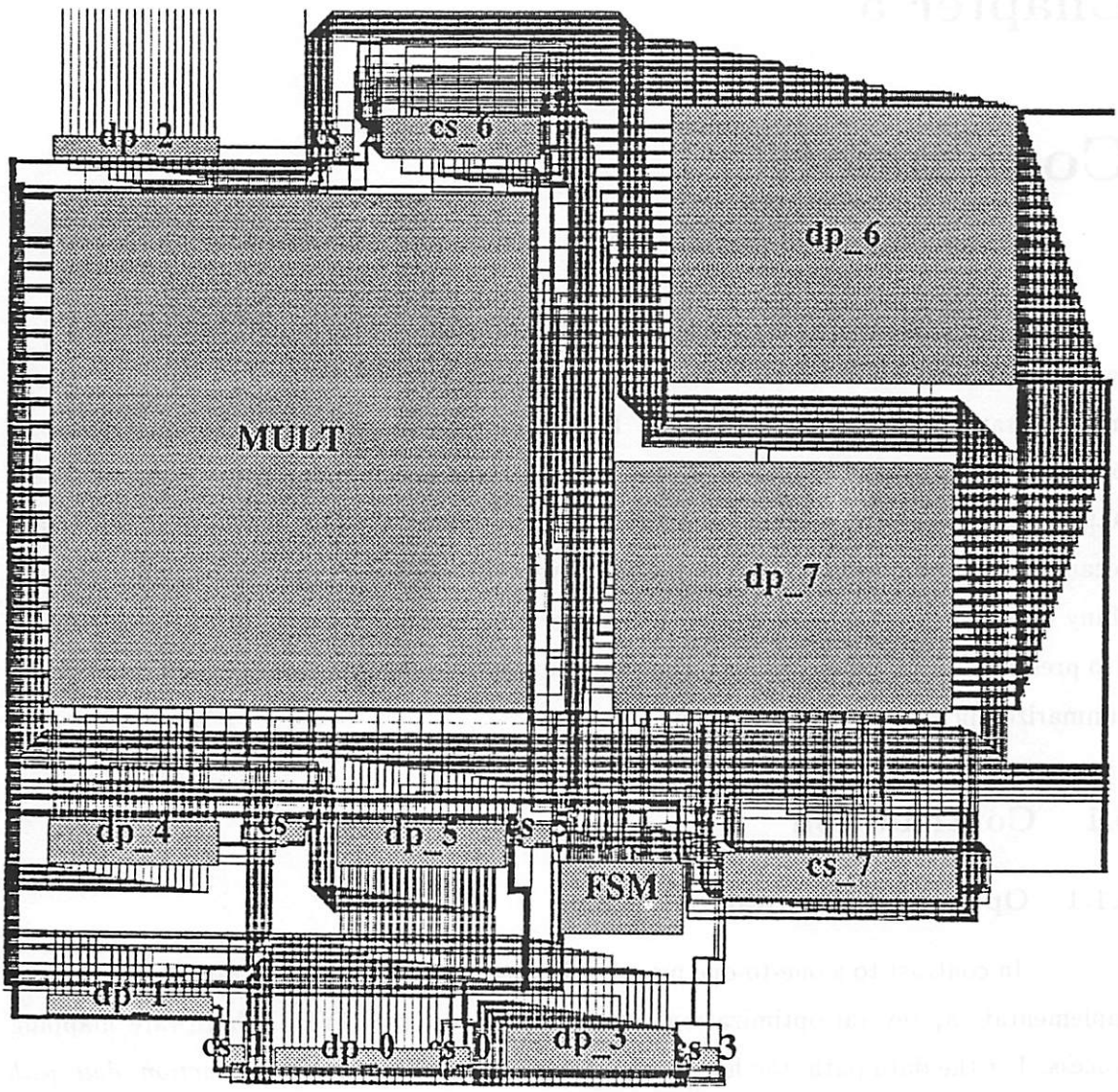

Figure 4.21: Layout of the 5th order WDF

Chapter 5

Conclusion

An interactive synthesis environment, HYPER, for high-performance real-time applications has been developed. This system has been successfully used to generate various implementations of several processors. In this project, we defined a hardware platform for the HYPER synthesis system, which involves three major components – the hardware mapper, the hardware selector, and the hardware database. This platform is important because it serves as the link between the high level synthesis system and the actual hardware. Many benchmarks experimenting with the operation and effectiveness of this platform are also presented in this dissertation. The contributions and future extensions of this work are summarized in the following sections.

5.1 Contribution

5.1.1 Optimized Hardware Mapping

In contrast to a one-to-one mapping between a decorated flow graph and its actual implementation, several optimization steps are taken in the HYPER hardware mapping process. For the data path, the hardware mapper performs *multiplexer reduction, data path partitioning, and register file merging.* For the control path, the mapper tries to *simplify the finite-state machine and the interface logic* by reducing the number of states and the control nets between different modules. Although these optimizations are mostly ad hoc, they play an extremely important role in generating efficient hardware. Through some comparisons with the manual design, the quality of these optimizations is demonstrated.

Various algorithms and heuristics have been developed for the optimizations. In developing these algorithms, we compared the quality of different approaches and chose the proper approach through the benchmark results. Furthermore, we studied the effect of the transformation order. The hardware mapper is gradually refined by this *generate-critique* process.

5.1.2 Clustering Based Hardware Selection

The HYPER hardware selection algorithm solves the clustering and the selection problems simultaneously. A proper clock rate will also be decided if not specified by the designer. This algorithm is able to handle *recursive, hierarchical, real-time graphs with timing constraints.*

A *ripple timing model*, which models the timing behavior of hardware modules at the block level, is proposed to facilitate the selection and clustering process. This model is based on operation chaining and uses sets of derivation rules to calculate the critical path. Timing and throughput constraints can be efficiently verified through this simple, yet accurate ripple model.

A *search mechanism* based on clustering and declustering is developed. Both the hardware selection problem and the clustering problem are combinatorial in nature. We therefore developed some heuristics, which employ the hardware utilization to direct the search, to reduce the search space.

An *accurate hardware cost estimation* is employed in the proposed algorithm. Since the hardware selection takes place before the allocation, assignment, scheduling and all the other synthesis steps in HYPER, a precise cost estimation is very difficult in the hardware selection. A hardware cost estimation, which is based on a relaxed-scheduling technique and reflects not only the execution unit cost but also the register cost, is therefore used to evaluate the quality of proposed solutions.

5.1.3 Hardware Database System

The information of the available hardware blocks is provided by a hardware database system, which contains information including delay, area, hardware parameters, and black-box views. Currently, the HYPER hardware database contains about thirty data-path modules, ten array modules, and ten standard-cell modules. New modules can be easily

introduced using the database editor, which is developed as part of the database system. The database system is important for both the hardware mapper and the hardware selector because it serves as an interface between the synthesis processes and the available cell library. A set of access routines have also been developed to facilitate the synthesis processes to search the database.

5.1.4 System Evaluation

Several processors for real-time applications, such as the Viterbi processor for connected-speech recognition and the 7th order IIR filter, have been designed and generated through HYPER to evaluate the system performance. The system is able to generate all the way down to layouts from the high level Silage descriptions. The quality of the HYPER design is then compared with that of the manual design to demonstrate the effectiveness of the synthesis system. The system evaluation is objective since the examples are real and the comparison is based on the layouts, rather than the estimation of the hardware costs.

Functional simulations of the designs have been performed using the Thor simulator to verify the correctness of the transformations in the synthesis process. Results from the functional simulation are checked against the results of the algorithmic simulation. Several problems on control macros, clocking strategies, initialization, and I/O have been detected and fixed based on the simulation results. This verification is extremely important for the synthesis system to demonstrate its usefulness.

With the HYPER synthesis system, tradeoffs of different designs can be easily made. Several implementations with different scheduling and allocation have been generated through HYPER to demonstrate the area-timing tradeoff of these implementations.

5.2 Future Work

Although many milestones have been reached in this project, many features, which are essential to make HYPER a complete and successful synthesis system, are still missing. Some of the features are research-related problems, such as the I/O management, the memory management, and the transformation mechanism; while the others are interface tools to make the system more user friendly or easier to adapt to other design styles. These features will be discussed in the following sections with the emphasis on the hardware mapping and the hardware selection.

5.2.1 Overall System

Future research directions of HYPER will focus on I/O, memory, and flow graph transformation issues. Various I/O protocols, both synchronous and asynchronous, should be supported. Different interprocessor communication schemes such as FIFO's and shared memories should be provided. These I/O options require the enhancement of the hardware database and some modifications of the current tools.

To achieve an efficient design, memory management is extremely important. The memory management problem decides where the memory should reside (on-chip or off-chip), how to partition the memory, what kind of memory to use (static or dynamic, single-ported or multiple-ported), and whether to use memory hierarchies. Solving the memory management problem requires extensive study and evaluation. Previous work on the memory management for DSP applications can be found in [30] and [31]; in which memory management strategies and several optimization tasks are proposed to compile multi-dimensional data structures into register files and SRAM's. Currently, HYPER assumes that arrays are stored in memory blocks and variables are stored in registers. This simple scheme still leaves much room for improvement and the previous work on the memory management can be a useful reference.

Although HYPER has already included many useful transformations such as retiming for synthesis, constant multiplication expansion, and pipelining, many more transformations are still needed to provide more design options or to make the design more efficient. An ideal transformation environment should provide not only a graphic front end and a set of transformation tools as most systems do, but also a transformation mechanism which helps the user in performing the transformations. We propose a transformation database which stores data including the conditions, the method, the effects/side-effects, and the expected results of a transformation based on estimation [59]. This database provides the designers with a clear picture of what kind of transformation can be performed and how to pursue such a transformation, and thus greatly simplifies the design task.

5.2.2 Hardware Mapping

Future work on hardware mapping will focus on the mapping of the decorated flow graphs onto different technologies or layout styles. The current hardware mapper can take any decorated flow graphs (with no assumptions on the hardware models) and map

them to a datapath-cluster architecture (Lager IV). Many optimizations are introduced in the mapper to improve the area utilization based on the customized, bit-sliced layout style and the register-file hardware model adopted by the HYPER scheduler. To interface with other layout styles, such as gate arrays or standard cells, a different set of transformations may be required. For example, data path partitioning, which tries to improve the layout efficiency for the bit-sliced data paths, may not be needed for the standard cell design since the place-and-route tool for the standard cells shall handle the partitioning problem automatically. However, a timing analyzer, which extracts the resistance and capacitance of routing wires and calculates the wire delay, may become crucial for the standard cell design. To decide a proper set of transformations for different design styles is not an easy task. The generate-critique process, as employed in the hardware mapper, should be used to achieve the layout efficiency.

In addition to the transformations for different design styles, some other utility tools are also very important for the hardware mapper. For example, a graphic tool which displays the register-transfer-level result from the synthesis process can help the designer easily understand the structural design. A more evolved database editor can help the designer introducing new cells without knowing the exact database format. Even though these tools do not involve hard research problems, they will make the system much more user-friendly.

5.2.3 Hardware Selection

The possible future work of hardware selection includes implementing other approaches such as the integer programming (or mixed integer programming) algorithm and the rule-based system and comparing the benchmark results of these approaches to find the best approach for the HYPER synthesis system. As described in Chapter 3, the performance of a hardware selection algorithm also depends on how well it interacts with the other synthesis processes. Therefore, the comparison should be based on the synthesized structure, rather than the hardware cost estimation. We have established the ripple model for delay evaluation and the relaxed-scheduling approach for cost estimation. These methodologies can also be used for the other approaches.

There is still much room left to improve the proposed clustering approach. The annealing schedule as well as the choices between the clustering move and the declustering

move can be further investigated. Some other cost estimations such as the stochastic min bound can also be tried out. In conclusion, the clustering and hardware selection problem is a hard problem. Although many interesting heuristics have been proposed, further evaluation is still needed. This evaluation may lead to a more evolved approach.

Bibliography

[1] *S/FILSYN Quick Reference Manual*, release 1.0 version 04 edition, April 1983.

[2] L. Hafer A. Parker. A formal method for the specification, analysis, and design of register-transfer level digital logic. *IEEE Transactions on CAD*, January 1983.

[3] B. Lin A. R. Newton. Synthesis of multiple level logic from symbolic high-level description languages. In *Proc. VLSI 89 Conference*, Munich, West Germany, August 1989.

[4] T. Villa A. Sangiovanni-Vincentelli. Nova: State assignment of finite state machines for optimal two-level logic implementations. In *26th ACM/IEEE Design Automation Conference Proceedings*, Las Vegas, June 1989. ACM/IEEE.

[5] D. G. Schweikert B. W. Kernighan. A proper model for the partitioning of electrical circuits. In *Proc. 9th Design Automation Workshop*, pages 298 - 301, 1972.

[6] Richard Blahut. *Fast Algorithms for Digital Signal Processing.* Addison-Wesley, 1985.

[7] R. W. Brodersen, editor. *A Framework and Tools for Silicon Compilation.* Kluwer Academic Publishers, Boston, MA., 1992.

[8] D. D. Gajski D. E. Thomas. Introduction to silicon compilation. In D. D. Gajski, editor, *Silicon Compilation*, chapter Chapter 1, pages 1 - 47. Addison-Wesley, 1988.

[9] R. A. Walker D. E. Thomas. Behavioral transformation for algorithmic level ic design. *IEEE Transactions on CAD*, 8(10):1115 - 1128, October 1989.

[10] D. Fogg. Operatior selection: Two approaches. In *Proceedings Forth High-Level Synthesis Workshop.* ACM. October 1989.

[11] L. Ramachandran D. Gajski. An algorithm for component selection in performance optimized scheduling. In *Proceedings IEEE ICCAD '91.* IEEE, November 1991.

[12] M. R. Garey D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[13] C. Tseng D. Siewiorek. Automated synthesis of data paths in digital systems. *IEEE Transactions on CAD*, 5(3):379 - 395, July 1986.

[14] D. W. Knapp. Feedback-driven datapath optimization in faslot. In *Proceedings IEEE ICCAD '90.* IEEE, November 1990.

[15] S. Devadas and A. R. Newton. Algorithms for hardware allocation in data path synthesis. In *Proc. IEEE ICCD Conference*, Cambridge, MA., October 1987. IEEE.

[16] G. Borriello E. Detjens. High-level synthesis: Current status and future directions. In *25th ACM/IEEE Design Automation Conference Proceedings*, pages 477 - 482. ACM/IEEE, July 1988.

[17] ERL, University of California at Berkeley. *Lager Tool Set*, December 1988.

[18] A. Stolzle et al. A flexible vlsi 60,000 word real-time continuous speechi recognition system. In *Proc. IEEE Workshop on VLSI Signal Processing*, San Diego, November 1990. IEEE.

[19] C. Chu et al. Hyper : An interactive synthesis environment for high performance real time applications. In *Proc. IEEE ICCD Conference*, Cambridge, MA., October 1989. IEEE.

[20] C. S. Shung et al. An intergrated cad system for algorithm-specific ic design. In *Proc. International Conference On System Design*, Hawaii, January 1989. IEEE.

[21] C. Tseng et al. A module binder for high level synthesis. In *Proc. IEEE ICCD Conference*, Cambridge, MA., October 1989. IEEE.

[22] D. E. Thomas et al. *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench.* Kluwer Academic Publishers, Boston, MA., 1990.

[23] D. Harrison et al. Data management and graphics editing in the berkeley design environment. In *Proc. IEEE ICCAD Conference*, Santa Clara, November 1986. IEEE.

[24] D. Lanneer et al. Architectural synthesis for medium and high throughput signal processing with the new cathedral environment. In R. Camposano and W. Wolf, editors, *High-Level VLSI Synthesis*, chapter Chapter 2, pages 27 – 54. Kluwer Academic Publishers, 1991.

[25] D. S. Johnson et al. Optimization by simulated annealing: An experimental evaluation; part i, graph partitioning. *Operations Research*, 37(6):865 – 892, November - December 1989.

[26] J. M. Rabaey et al. Cathedral-ii: A synthesis system for multiprocessor dsp systems. In D. D. Gajski, editor, *Silicon Compilation*, chapter Chapter 8, pages 311 – 360. Addison-Wesley, 1988.

[27] J. M. Rabaey et al. A large vocabulary real time continuous speech recognition system. In R. Brodersen H. Moscovitz, editor, *VLSI Signal Processing*. IEEE Press, 1988.

[28] J. M. Rabaey et al. *HYPER v1.0.* University of California, Berkeley, January 1991.

[29] J. Rabaey et al. Fast prototyping of datapath-intensive architectures. In *IEEE Design and Test of Computers*, June 1991.

[30] J. Vanhoof et al. Compiling multi-dimensional data streams into distributed dsp asic memory. In *Proceedings IEEE ICCAD '91*. IEEE, November 1991.

[31] P. E. R. Lippens et al. Memory synthesis for high speed dsp applications. In *Proceedings IEEE CICC '91*, San Diego, CA., May 1991. IEEE.

[32] R. Alverson et al. *THOR User's Manual.* Stanford University, January 1988. Tech. Rep. CSL-TR-88-348 and 349.

[33] R. Bergamaschi et al. Data-path synthesis using path analysis. In *28th ACM/IEEE Design Automation Conference Proceedings*, San Francisco, June 1991. ACM/IEEE.

[34] R. Brayton et al. Mis: A multiple-level logic optimization system. *IEEE Transactions on CAD*, 6(6):1062 – 1081, November 1987.

[35] R. Jain et al. Custom design of a vlsi pcm-fdm transmultiplexer from system specifications to circuit layout using a computer-aided design system. *IEEE JSSC*, SC-21(1):73 – 85, February 1986.

[36] R. Jain et al. Module selection for pipelined synthesis. In *25th ACM/IEEE Design Automation Conference Proceedings.* ACM/IEEE, June 1988.

[37] R. Jain et al. Experience with the adam synthesis system. In *26th ACM/IEEE Design Automation Conference Proceedings*, Las Vegas, June 1989. ACM/IEEE.

[38] S. Kirkpatrik et al. Optimization by simulated annealing. *Science*, 220(4598):671 - 680, May 1983.

[39] S. Note et al. Automated synthesis of a high speed cordic algorithm with the catehdral-iii compilation system. In *Proceedings ISCAS '88*, Helsinki, 1988. IEEE.

[40] S. Note et al. Combined hardware selection and pipelining in high performance data-path design. In *Proceedings IEEE ICCD '90*, Cambridge, MA., September 1990. IEEE.

[41] F. I. Romeo. *Simulated Annealing: Theory and Applications to Layout Problems.* PhD thesis, University of California, Berkeley, March 1989.

[42] F. S. Roberts. *Applied Combinatorics.* Prentice-Hall, 1984.

[43] R. Brayton E. Sentovich F. Somenzi. Don't cares and flobal flow analysis of boolean networks. In *Proceedings IEEE ICCAD '88.* IEEE, November 1988.

[44] C. M. Fiduccia and R. Mattheyses. A linear-time heuristic for improving network partitions. In *19th ACM/IEEE Design Automation Conference Proceedings.* ACM/IEEE, July 1982.

[45] G. W. Leive. *The Design, Implementation, and Analysis of an Automated Logic Synthesis and Module Selection System.* PhD thesis, Carnegie-Mellon University, January 1981.

[46] L. Gazsi. Explicit formulas for lattice wave digital filters. *IEEE Transactions on Circuits and Systems*, CAS-32(1):68 - 88, January 1985.

[47] Y. S. Foo H. Kobayashi. A knowledge-based system for vlsi module selection. In *Proceedings IEEE ICCAD '86.* IEEE, November 1986.

[48] M. Potkonjak J. M. Rabaey. Retiming for scheduling. In *Proc. IEEE Workshop on VLSI Signal Processing*, San Diego, November 1990. IEEE.

[49] P. G. Paulin J. P. Knight. Force-directed scheduling for the behavioral synthesis of asic's. *IEEE Transactions on CAD*, pages 661 - 679, June 1989.

[50] M. Potkonjak J. Rabaey. A scheduling and resource allocation algorithm for hierarchical signal flow graphs. In *26th ACM/IEEE Design Automation Conference Proceedings*, Las Vegas, June 1989. ACM/IEEE.

[51] N. Weste K. Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*, pages 196 - 201. Addison Wesley, 1985.

[52] J. W. Greene K. J. Supowit. Simulated annealing without rejected moves. *IEEE Transactions on CAD*, 5(1):221 - 228, January 1986.

[53] S. K. Parker K. W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1191 - 1201, 1988.

[54] U. Kaiser. Wave digital filters and their significance for customized digital signal processing. *TI Engineering Journal*, pages 29 - 44, September - October 1985.

[55] C. Kring and A. R. Newton. A cell-replication approach to mincut-based circuit partitioning. In *Proceedings IEEE ICCAD '91.* IEEE, November 1991.

[56] J. S. Lis and D. D. Gajski. Vhdl synthesis using structured modeling. In *26th ACM/IEEE Design Automation Conference Proceedings.* ACM/IEEE, June 1989.

[57] M. McFarland. Using bottom-up design techniques in the synthesis of digital hardware from abstract behavioral descriptions. In *23th ACM/IEEE Design Automation Conference Proceedings.* ACM/IEEE, June 1986.

[58] M. Potkonjak. Hierarchical probabilistic graph partitioning algorithm. Intrnal Report, University of California, Berkeley, 1989.

[59] J. Rabaey M. Potkonjak. Resource driven synthesis in the hyper system. In *Proc. IEEE ISCAS Conference*, New Orleans, May 1990. IEEE.

[60] P. Hilfinger. A high level language and silicon compiler for digital signal processing. In *Proc. IEEE Custom Integrated Circuits Conference.* IEEE, May 1985.

[61] P. Hilfinger. *Silage: A Language for Signal Processing.* University of California, Berkeley, March 1989.

[62] Miodrag Potkonjak. *High Level Synthesis: Resource Utilization Approach.* PhD thesis, University of California, Berkeley, 1991.

[63] P. Reutz R. Brodersen. A realtime image processing chip set. In *Proceedings International Solid State Circuit Conference*, pages 148 - 149, San Francisco, February 1986. IEEE.

[64] R. Yu D. Chen J. Rabaey R. Brodersen. A vlsi grammar processing subsystem for a real-time large vocabulary continuous speech recognition system. In *Proceedings IEEE CICC '90*, Boston, MA., May 1990. IEEE.

[65] M. McFarland A. C. Parker R. Camposano. Tutorial on high-level synthesis. In *25th ACM/IEEE Design Automation Conference Proceedings.* ACM/IEEE, June 1988.

[66] R. Jain. Mosp: Module selection for pipelined designs with multi-cycle operations. In *Proceedings IEEE ICCAD '90*, Santa Clara, November 1990. IEEE.

[67] D. C. Chen R. Yu J. Rabaey R. W. Brodersen. A vlsi grammar procseeing subsystem for a real-time large-vocabulary continuous speech recognition system. *IEEE Journal of Solid-State Circuits*, 26(3):443 - 448, March 1991.

[68] John Riordan. *An Introduction to Combinatorial Analysis.* Princeton University Press, 1980.

[69] B. W. Kernighan S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Tech. Journal 49:2*, pages 291 - 307, February 1970.

[70] M. T. Trick S. W. Director. Lassie: Structure to layout for behavior synthesis tools. In *26th ACM/IEEE Design Automation Conference Proceedings*, Las Vegas, June 1989. ACM/IEEE.

[71] Robert Sedgewick. *Algorithms.* Addison Wesley, 1983.

[72] R. B. Segal. *BDSYN Users' Manual, Version 1.1.* University of California, Berkeley, September 1987.

[73] J E Volder. The cordic trigonometric computing technique. *IRE Transactions on Electronic Computers*, pages 330 - 334, 1959.

[74] R. Walker. A survey of high-level synthesis systems (second edition). Technical Report 90-30, Rensselaer Polytechnic Institute, October 1990.

[75] J S Walther. A unified algorithm for elementary functions. In *Proceedings of the 1971 Spring Joint Computer Conference*, pages 379 – 385. IEEE, IEEE, 1971.

Appendix A

User's Manual

This appendix contains the information on how to use the hardware mapper and the hardware selector in HYPER. Both the hardware mapper and the hardware selector can be fired through xhyper – the synthesis manager of HYPER under X-window, or independently.

A.1 Hardware Mapper

HyperMap -- Hardware mapper for Hyper

SYNOPSIS

HyperMap [-R] [-p] [-n] [-s] [-v] *flowgraph*

DESCRIPTION

HyperMap takes a Lisp format flowgraph description (usually generated by **Flow2Lisp**) as the input and produces all the necessary files to generate layouts through **LAGER IV**. Hyper assumes a datapath cluster architecture; therefore, the **LAGER** files generated include:

(1) sdl files for datapaths, which are called dp_#.sdl. # is 0, 1, 2 etc.

(2) sdl files and bds files for control slices, which are called cs_dp_#.sdl and cs_dp_#.bds respectively. Again, # is a number starting from 0. cs_dp_0 matches with dp_0, cs_dp_1 matches with dp_1, etc. Control slices will be implemented by standard cells.

(3) A bds file for the central control, which is called FSM1.bds. The central control will be implemented by a finite state machine.

(4) A sdl file and a parval file of the highest hierarchy, which are called processor.sdl and processor.parval respectively.

(5) A log file, processor.log, which records some information for debugging.

(6) A file called processor.sta which contains some statistics of the processor.

(7) An optional file, ck_gen.sdl, for buffering clock signals. This module will be implemented by standard cells.

The hardware mapper needs both the flowgraph description and the database information of available cells. The flowgraph description is a Lisp program which will be compiled in the mapping process. It can be generated by **Flow2Lisp** from the ASCII flowgraph format or hand written by a designer. The detail format of the description are explained in Appendix C. The cell databases are also in the Lisp format. The default databases are ~hyper/lib/Hardware/rb-dp, which is the datapath cell library, and ~hyper/lib/Hardware/rb-array, which is the memory block and array module library. For more information on the database, see Appendix B.

HyperMap starts up the Lisp environment (acl), loads the sources files, compiles the flowgraph description and performs transformations and optimizations on the flowgraph. The major steps of the mapping process and some statistics of the processor will be displayed as the process goes on. These information helps the user to find out the complexity of a design and the possible reasons of failures in the mapping process.

OPTIONS

-R Specify the path of the hardware database library. The default path is ~hyper/lib/Hardware.

-p Specify if datapath partition is needed. The default is t (true) for performing the partition. If no partition is needed, the value should be given nil.

-n Specify the number of datapaths after partitioning. This is only used if -p flag is t.

-s Number to limit the largest datapath size. This number is only used if -p flag is t and -n flag is not specified. The default value of this flag is 120,000, which is about 20 blocks in a datapath.

-v Verbose mode

FILES

~hyper/lib/Hardware/rb-dp
~hyper/lib/Hardware/rb-array

SEE ALSO

Flow2Lisp

AUTHORS

Chi-Min Chu
University of California, Berkeley
chu@zabriskie.Berkeley.EDU

BUGS

In case of an error, read processor.log to find out more information. If some of the error messages are not understandable, users can use :zoom command in the acl environment to find out what causes the error. For the datapath partitioning, if the largest datapath size (-s flag) is given too small (less than a block) or the number of the datapaths after partitioning (-n flag) is too large, the program may ran into an infinite loop. Even though the hardware mapper has performed many optimizations in generating efficient LAGER layouts, more optimization routines can be introduced in the future.

A.2 Hardware Selector

NAME

hwSelect -- Hardware Selection and Register Assignment Module

SYNOPSIS

hwSelect [-a] [-A] [-c] [-s] [-d] [-v] [-p] *flowgraph*

DESCRIPTION

Given a behavior description of an algorithm represented by a signal flow graph, the tasks of hardware selection are to decide the length of clock cycles (if not specified by the user), to choose proper hardware blocks for the nodes, and to assign certain edges to be registers so that the minimized hardware cost under the timing and throughput constraints can be achieved. This program uses the path specified by the *hyper* file to search for the hardware databases for choosing proper hardware modules.

The best solution found to satisfy the timing and throughput constraints will be back annotated onto the flowgraph, with the clock cycle, the register assignment, and the hardware chosen annotated as appropriate attributes. Notice that hwSelect may produce MACRO nodes in the flowgraph if hardware functions are created in this phase. If no solution is found, the program aborts with appropriate error messages and no output file is produced.

OPTIONS

- -a Loading the input from an afl-database. The default is input from an OCT database.
- -A Storing the output to an afl-database. The default is output to an OCT database.
- -d Debug mode.
- -c Specifying the length of a clock cycle in nsec.
- -s Specifying the sampling period (ie. the input sample rate) of the algorithm in nsec. This number determines the throughput.
- -p Fully pipelined structure, no clustering is performed. The throughput constraint may be violated in this case.
- -v Verbose mode. Displays various characteristics regarding the hardware selection process and its results.

FILES

~hyper/lib/hyper
~hyper/lib/Hardware/rb-dp
~hyper/lib/Hardware/rb-array

AUTHORS

Chi-Min Chu
University of California, Berkeley
chu@zabriskie.berkeley.edu

BUGS

This program provides minimal processing of the read/write nodes. All the read/writes nodes will be annotated as the RAM modules and registers are allocated for the I/O ports of the memories.

Appendix B

HYPER Hardware Database Format

B.1 Introduction

This document describes the hardware database format for the HYPER synthesis environment. This format is in the Lisp syntax and it is a hierarchical database. The author would assume that readers have the basic Common Lisp background in reading this document. The best way to understand the database is by going through a complete example. An example of this database can be found in Section 4 of this Appendix, which is a database currently used for the data path synthesis. Another example can be found in Section 5, which is the database currently used for the memory and array modules. A subset of the database for the standard cells are also listed in this appendix and it can be found in Section 3. The structures of these three databases are the same, but the information required is slightly different for each case. In the following sections, the general structure of the databases will be outlined first, followed by the detailed format of each database.

All the three databases described in this appendix are based on the Lager IV cell library and have been used in many real examples including a 7th order IIR filter, the Epsilon processor, the Viterbi processor etc. Although many attributes of the hardware modules have been incorporated in the database, more features such as testing capabilities and power consumption information of the hardware modules may still be needed in the future.

The reason for designing the database in the Lisp format instead of using OCT is for the ease of modification. Different strategies on the database have been tested without working on OCT to speed up the modification. When the final strategy is determined, the database should be rewritten in OCT or in another object oriented environment. Questions and comments on the database format can be mailed to chu@zabriskie.berkeley.edu.

B.2 Structure of Database

The structure of the database can be illustrated in Figure B.1. At the top level, function names are used as the key. The database returns a list of cells that implement the function. Then the cellname is used to access the information of the cell under this function. Information of a cell is slightly different for the three databases. For the data path cell library, information of a cell includes parameters, area, delay, one-bit-delay, ripple-delay, ripple-offset, data-terminal, power-terminal, ctl-in-terminal, ctl-out-terminal, ctl-term-edge, complement-out, and driving-cap. For the array cell library, information of a block contains parameters, area, delay, one-bit-delay, ripple-delay, ripple-offset, data-terminal, power-terminal, ctl-in-terminal, complement-out, reducibility and driving-cap. For the standard cell library, only area, delay, data-terminal, complement-out, and driving-cap are needed. We can see that the data path cell library has the most information, memory block cell library has all the information except ctl-out-terminal and ctl-term-edge. The standard cell database has the least information and is a subset of the other two. The data format of each item in various databases may be different and will be further described in the following sections.

The databases are organized as the Common Lisp associate lists. To be precise, The data are of the following format:

```
database :=
    ((function1
        (cell-name1
                  (item-name1 data1)
                  (item-name2 data2)
                   ;; ...
                   )
        (cell-name2
```

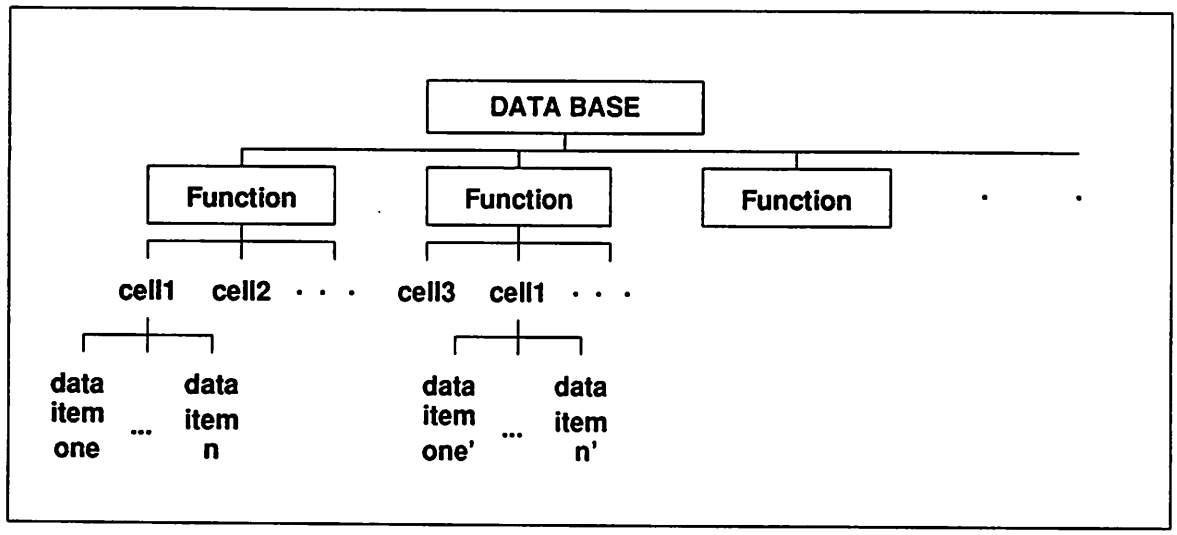

Figure B.1: Structure of the Database

```
                    ;; ...)
        ;; ...)
(function2
        ;; ...
 )
...)
```

If a cell can perform multiple functions, it may occur in the database several times. The wiring and delay information of the cell may be different for different cases to perform various functions.

A database editor which helps the designer interfacing with the database system has been implemented. It works under the Lisp environment. Several access functions, input functions, and delete functions are available to the users so that they do not have to remember the detailed structure of the database. Nevertheless, the users still need to know the data format for each item. The default value for each data item is nil. That is, if a data item is not specified, "nil" will be assumed.

Attribute	Format	Example	Note
area	s-exp	(* 70 100)	in lambda square
delay	s-exp	2	in nsec
data-terminal	(out (in1 in2 ..))	(O2 (1A 1B))	
complement-out	symbol	O1	possible values: nil/term-name
driving-cal	symbol	no	possible values: small/big/no

Table B.1: Data Item and Format of Standard Cell Database.

B.3 Standard Cell Database

This section describes the simplest database – standard cell database. Each standard cell contains five data items. Table B.1 lists these items and data formats.

Standard cells are mainly used for glue logic between data paths and central control (FSM). Therefore, the functions of the standard cells in the HYPER system are purely combinational and the attributes required in the database are very easy. Both the area and delay of a cell are s-expressions to represent the size and the critical delay of the cell. The data-terminal attribute specifies the input/output terminals. The output terminal is a single terminal which appears as the first element of the attribute (which is a list). The input terminals are an ordered list, which is the second element of the attribute. Complement-out specifies if a complementary output terminal is available in the cells. If there is a complementary output terminal, the terminal name is given in the attribute. Driving-cap specifies the driving capability of the cell. The hardware mapper is able to choose a cell with a proper driving capability using this value. Figure B.2 is a sample standard cell library which shows some simple standard cells in the current Lager library and gives the flavor of the database format.

B.4 Database for Data Path Modules

The database for the data path modules has more attributes than the standard cell database. These attributes are listed as follows with a description of their formats:

parameters This attribute specifies the hardware parameters of a data path block. The format of the parameters attribute is a list of parameters. For most data path blocks, this list contains only one element - N (the word width of the block). For more

```
((and ("nanf211"
       (AREA (* 76 38)) (DELAY 2)
       (DATA-TERMINAL (O2 (A1 B1)))
       (COMPLEMENT-OUT O1) (DRIVING-CAP NO)))
  (or ("norf211"
       (AREA (* 76 46)) (DELAY 2)
       (DATA-TERMINAL (O1 (A1 B1)))
       (COMPLEMENT-OUT O2) (DRIVING-CAP NO)))
  (not ("invf101"
        (AREA (* 76 22)) (DELAY 1)
        (DATA-TERMINAL (O (A1)))
        (COMPLEMENT-OUT nil) (DRIVING-CAP NO))
       ("invf103"
        (AREA (* 76 38)) (DELAY 1)
        (DATA-TERMINAL (O (A1)))
        (COMPLEMENT-OUT nil) (DRIVING-CAP SMALL))
       ("invf104"
        (AREA (* 76 46)) (DELAY 1)
        (DATA-TERMINAL (O (A1)))
        (COMPLEMENT-OUT nil) (DRIVING-CAP BIG)))
  (xor ("xorf201"
        (AREA (* 76 54)) (DELAY 2)
        (DATA-TERMINAL (O1 (A1 B1)))
        (COMPLEMENT-OUT nil) (DRIVING-CAP NO)))
  (buffer ("buff101"
           (AREA (* 76 30)) (DELAY 1)
           (DATA-TERMINAL (O (A1)))
           (COMPLEMENT-OUT nil) (DRIVING-CAP BIG)))
 )
 ;; (1) Delay is based 2u technology, nominal case, primary output.
 ;; (2) Since std cells are used for glue logic, only the following
 ;;     information is needed :
 ;;     (A) area, delay
 ;;     (B) data-terminal, complement-out
 ;;     (C) driving-cap
 ;; (3) If sequential logic is going to be built, the control terminals
 ;;     will be needed. The timing information can be managed as in
 ;;     rb-dp.
```

Figure B.2: Sample Standard Cell Library

Name	Description
N	output word width
M	number of bits shift
CONSTANT	constant to compare with
PATTERN	bit string that forms the constant block

Table B.2: Keyword in the Parameters Attribute of Data Path Database.

Name	Parameters	Description
structure_left	M	decide log shifter structure from # bits shifted
structure_right	M	same as shift_left except for shifting right

Table B.3: Functions Defined for the Parameters Attribute of Data Path Database.

complicated blocks, parameters may contain more than one hardware parameter. For example, Log shifters have two hardware parameters: N and M (the number of the shifted bits of the shifters). The element of the parameter list can take two forms: It can either be a list of ($parameter_i$ $keyword_i$) or it can be simply $parameter_i$, representing the keyword is the same as the parameter name $parameter_i$.

Some of the hardware parameters can be calculated from the attributes or arguments of the decorated flow graph. Some of them, on the other hand, can only be decided when silicon compilation is performed. For the hardware parameters that cannot be calculated from the flow graph parameters, their $keyword_i$ should be USER_DEFINE. For the other parameters, their $keyword_i$ should be Lisp expressions. A Lisp expression can either be a keyword or a function of the keywords which are recognized by the database. A list of internally defined keywords (or called flow graph parameters) and internally defined functions specifically used for this attribute are shown in Table B.2 and Table B.3 respectively. A list of general Lisp functions that are used for both the parameters attribute and the other s-expression attributes are listed in Table B.4. More functions can be included in the list in the future to expand the HYPER feature.

range This attribute specifies the shift range of a shifter. The format of the range attribute is ($lower-bound$ $upper-bound$), where $lower-bound$ and $upper-bound$ are two

Name	Parameters	Description/Example
+	N1, N2 ...	N1 + N2 + ...
-	N1, N2 ...	N1 - N2 - ...
*	N1, N2 ...	N1 * N2 * ...
/	N1, N2 ...	(N1 / N2) / ...
sqrt	N	square root of N
exp	N	e^N
expt	N1, N2	$N1^{N2}$
log	N1, N2	$log_{N2}N1$
ceiling	N1	$\lceil N1 \rceil$
length	L1	length of list L1
my-length	L1, L2	(my-length '(1 3) '(a b c)) = (a c)

Table B.4: Lisp Functions Used for All Attributes of Hardware Database

integers to represent the maximal number and the minimal number of bits shifting (can be negative meaning shifting toward the opposite direction).

area This attribute specifies the area of a module. It is a Lisp expression of the hardware parameters of the module.

delay This attribute specifies the critical delay of a module and has the same format as the area attribute, a function of the hardware parameters of the module. Currently, the values of the delay attributes are based on the 2-micron CMOS technology and represent the nominal case delay of the modules. The delay attribute may be expanded in the future to include delay values of various technologies, worst case delays, and delays for different loads. A file of delay values may be needed for each module in this case.

one-bit-delay This attribute is used in the ripple model to specify the one bit delay of the module. It is a Lisp expression of hardware parameters.

ripple-delay Same as one-bit-delay except that this attribute specifies the ripple delay of the module. If the ripple delay has a positive value, the ripple direction of the module is LSB to MSB (ripple to the left). On the other hand, if the ripple delay is negative, the ripple direction of the module is MSB to LSB (ripple to the right).

ripple-offset This attribute specifies the ripple offset of a module used in the ripple model. Similar to the ripple-delay attribute, the signs of the ripple-offset values indicate the ripple-offset directions.

data-terminal This attribute specifies the data terminal connection of the module. The format is (*output−terminal* (*input−terminal*$_1$ *input−terminal*$_2$ ...)) where *output−terminal* is the name of the primary output terminal and *input − terminal*$_i$'s are the names of the input terminals. Input terminals have to be listed according to the input terminal order of the performed function. This order can be found in Appendix C for all the functions recognized by the hardware mapper.

power-terminal This attribute specifies which power terminals have to be connected for the module. It's a list of two possible elements: Vdd (power) and GND (ground).

ctl-in-terminal This attribute specifies the required control input connection for the module. The format is a list of elements. Each element is either "(terminal-name connection)" or simply "terminal-name", meaning the terminal name is the same as the connection. Connection can be a keyword or an expression of the recognized *keywords* (not an expression of the control terminals of the module). Table B.5 is a list of expressions recognized by the hardware mapper. In addition to Boolean functions, the expression can also be an *if* function. The format of the if function is *(if condition expression1 expression2)*, meaning that if *condition* is true, the control input is expression1, otherwise, the control input is expression2. This feature is very useful for modules such as adders and counters, in which the odd bit-slice is different from the even bit-slice and therefore different control connections are required for modules of even and odd bit-widths. For this specific application, *condition* should be either oddp (meaning odd number of bits) or evenp (meaning even number of bits).

The keywords used in the ctl-in-terminal attribute is listed in Table B.6. Some of the keywords may only be recognized by a certain modules performing a specific function, while the others are general keywords and are recognized by the hardware mapper for all modules. Notice that control input terminals of one module such as an ALU may be connected differently to perform different functions. Another important feature of the hardware database is that control buses are supported. This feature is mainly used for modules such as log shifters which have a decoded/encoded bus as the control

Function	Parameters	Description
and	B1, B2, ...	B1 and B2 ...
or	B1, B2, ...	B1 or B2 ...
not	B1	not B1
nand	B1, B2, ...	B1 nand B2 ...
nor	B1, B2, ...	B1 nor B2 ...
xor	B1, B2, ...	B1 xor B2 ...

Table B.5: Boolean Functions in the Hardware Database.

input terminals. The format for the control bus is as follows:

```
(ctl-term-name BUS (bus-width bus-connection))
```

where BUS is a keyword to specify that ctl-term-name is a control bus. Both bus-width and bus-connection are Lisp expressions to specify the control bus width and the control bus connection. Two functions are defined specifically for the bus connection of the log shifters: encoded_SHIFT and decoded_SHIFT. Both functions take one parameter as input: the number of bits shifted of the log shifter. Encoded_SHIFT specifies that the control input bus is decoded and hence an encoder is needed. decoded_SHIFT, on the other hand, specifies that no encoder is needed for the control bus connection.

ctl-out-terminal This attribute specifies the connections of the control output terminals. If a module does not have any control output, this attribute should be nil. Otherwise, this attribute can either be a control terminal name, meaning this terminal is the primary control output, or a Lisp expression specifying the required boolean functions in the control slice of the module. For the latter case, the basic format is ($function$ $terminal_1$ $terminal_2$ ...) where $function$ is a boolean function listed in Table B.5 and $terminal_i$'s are the control terminal names. The *if* function described in the control-in-terminal attribute is also available for this attribute. Table B.7 lists the primary control output for each function. For functions not listed in the Table, their primary output should be nil.

ctl-term-edge This attribute specifies the terminal edges of control terminals. If the ctl-term-edge attribute of a control terminal is not specified, this terminal may be brought

Keyword	Recognized by Which Function	Description/Meaning
Vdd (or 1)	all	connect to high
GND (or 0)	all	connect to low
CK1	all	connect to clock signal one
CK2	all	connect to clock signal two
LOAD	register, counter, memory	load input
OEN	register, counter, memory	output enable
R/W	memory	1 to read memory, 0 to write memory
SHIFT	shifter	1 to perform shift, 0 otherwise
COUNT	counter	1 to perform count, 0 otherwise
SELECT	mux	1 to select input1, 0 to select input2
RESET	register, counter	1 to reset the storage, 0 otherwise
KEEP	scan register	1 to keep the scan value, 0 otherwise
SCAN	scan register	1 to scan input, 0 otherwise
SHIFT_IN_NUM	shifter	number to shift in: 1/0/sign
L#	shifter	left shift # bits
R#	shifter	right shift # bits

Table B.6: Keyword for Specifying CTL-IN-TERMINAL Attribute.

Function	Primary Control Output
+, - , ++	carry out
==, !=, >=, <=, >, <	comaprison result

Table B.7: Primary Control Output of Functions.

out either side (top or bottom). The format of this attribute is a list in which each element is of the from: "(control-terminal-name top/bottom)".

complement-out This attribute, as in the standard cell library, specifies if a complementary output terminal is available.

driving-cap This attribute is the same as that in the standard cell library to specify the driving capability of the module.

The following subsection is a print-out of the current hardware database of the data path modules in HYPER. Most of the cells are from the Lager cell library. Some of them, however, are dummy cells for the demo purpose or for the ease of the synthesis process. When reading this print-out, keep in mind that the database is read by the Lisp

interpreter and that the interpreter does not distinguish lower-case and upper-case letters. The default of the interpreter is upper-case. Therefore, if lower-case letters such as the module names are needed, quotes have to be used.

B.4.1 Database of Data Path Modules in HYPER

```
((++ ("counterO" (PARAMETERS (N)) (AREA (* N 263 (/ (+ 50 51) 2)))
                 (DELAY (+ 5 (* N 3)))
                 (ONE-BIT-DELAY 5)
                 (RIPPLE-DELAY (* N 3))
                 (RIPPLE-OFFSET 0)
                 (DATA-TERMINAL (OUT (IN)))
                 (POWER-TERMINAL (Vdd GND))
                 (CTL-IN-TERMINAL
                  ((PHIA CK1) (PHIAINV (NOT CK1)) (PHIB CK2)
                   (PHIBINV (NOT CK2)) (CIN GND) (LOAD LOAD)
                   (LOADINV (NOT LOAD))
                   ;              ^^^^ keyword,
                   (COUNT COUNT) (COUNTINV (NOT COUNT))))
                 (CTL-OUT-TERMINAL COUTINV)
                 (CTL-TERM-EDGE
                  ((PHIA TOP) (PHIAINV TOP) (PHIB TOP) (PHIBINV TOP)
                   (CIN BOTTOM) (LOAD TOP) (LOADINV TOP) (COUNT TOP)
                   (COUNTINV TOP) (COUTINV TOP)))
                 (COMPLEMENT-OUT OUTINV) (DRIVING-CAP NO))
    ("counterE" (PARAMETERS (N)) (AREA (* N 263 (/ (+ 50 51) 2)))
                 (DELAY (+ 5 (* N 3)))
                 (ONE-BIT-DELAY 5)
                 (RIPPLE-DELAY (* N 3))
                 (RIPPLE-OFFSET 0)
                 (DATA-TERMINAL (OUT (IN)))
                 (POWER-TERMINAL (Vdd GND))
                 (CTL-IN-TERMINAL
                  ((PHIA CK1) (PHIAINV (NOT CK1)) (PHIB CK2)
                   (PHIBINV (NOT CK2)) (CIN GND) (LOAD LOAD)
                   (LOADINV (not LOAD))
                   (COUNT COUNT) (COUNTINV (NOT COUNT))))
                 (CTL-OUT-TERMINAL COUT)
                 (CTL-TERM-EDGE
                  ((PHIA TOP) (PHIAINV TOP) (PHIB TOP) (PHIBINV TOP)
                   (CIN BOTTOM) (LOAD TOP) (LOADINV TOP) (COUNT TOP)
                   (COUNTINV TOP) (COUT TOP)))
                 (COMPLEMENT-OUT OUTINV) (DRIVING-CAP NO)))
 (>= ("comparatorE" (PARAMETERS (N)) (AREA (* N 211 49))
                    (DELAY (* N 2))
                    (ONE-BIT-DELAY 0)
                    (RIPPLE-DELAY (- (* N 2)))
                    (RIPPLE-OFFSET 0)
                    (DATA-TERMINAL (NIL (A B)))
                    (POWER-TERMINAL (Vdd GND))
                    (CTL-IN-TERMINAL
                     ((AGTBININV VDD) (AEQBIN VDD)))
                    (CTL-OUT-TERMINAL (not AGTBINV))
                    (CTL-TERM-EDGE
                     ((AGTBININV TOP) (AGTBINV BOTTOM)))
                    (COMPLEMENT-OUT NIL) (DRIVING-CAP NO)))
```

```
(== ("compconst" (PARAMETERS (N CONSTANT)) (AREA (* N 45 47))
                 (DELAY (* N 1))
                 (ONE-BIT-DELAY 0)
                 (RIPPLE-DELAY (- (* N 1)))
                 (RIPPLE-OFFSET 0)
                 (DATA-TERMINAL (NIL (IN)))
                 (POWER-TERMINAL (Vdd GND))
                 (CTL-IN-TERMINAL NIL)
                 (CTL-OUT-TERMINAL CNTOUT)
                 (CTL-TERM-EDGE ((CNTOUT TOP)))
                 (COMPLEMENT-OUT NIL) (DRIVING-CAP NO))
    ("comparatorE" (PARAMETERS (N)) (AREA (* N 211 49))
                   (DELAY (* N 2))
                   (ONE-BIT-DELAY 0)
                   (RIPPLE-DELAY (- (* N 2)))
                   (RIPPLE-OFFSET 0)
                   (DATA-TERMINAL (NIL (A B)))
                   (POWER-TERMINAL (Vdd GND))
                   (CTL-IN-TERMINAL
                    ((AEQBIN VDD) (AGTBININV VDD)))
                   (CTL-OUT-TERMINAL AEQB)
                   (CTL-TERM-EDGE
                    ((AEQBIN TOP) (AEQB BOTTOM)))
                   (COMPLEMENT-OUT NIL) (DRIVING-CAP NO)))
(<= ("comparatorE" (PARAMETERS (N)) (AREA (* N 211 49))
                   (DELAY (* N 2))
                   (ONE-BIT-DELAY 0)
                   (RIPPLE-DELAY (- (* N 2)))
                   (RIPPLE-OFFSET 0)
                   (DATA-TERMINAL (NIL (A B)))
                   (POWER-TERMINAL (Vdd GND))
                   (CTL-IN-TERMINAL
                    ((AEQBIN VDD) (AGTBININV VDD)))
                   (CTL-OUT-TERMINAL (OR AEQB AGTBINV))
                   ;      terminal names ^^^^ ^^^^^^^
                   (CTL-TERM-EDGE
                    ((AEQBIN TOP) (AGTBININV TOP) (AEQB BOTTOM)
                     (AGTBINV BOTTOM)))
                   (COMPLEMENT-OUT NIL) (DRIVING-CAP NO)))
(REG ("register" (PARAMETERS (N)) (AREA (* N 47 105))
                 (DELAY 2)
                 (ONE-BIT-DELAY 2)
                 (RIPPLE-DELAY 0)
                 (RIPPLE-OFFSET 0)
                 (POWER-TERMINAL (Vdd GND))
                 (DATA-TERMINAL (DATABUS (DATABUS)))
                 (CTL-IN-TERMINAL
                  ((LOAD (AND LOAD CK1))
                   (LOADINV (NOT (AND LOAD CK1)))
                   (OEN (AND CK2 OEN))
                   (OENINV (NOT (AND OEN CK2)))))
                 (CTL-OUT-TERMINAL NIL)
                 (CTL-TERM-EDGE
                  ((LOAD TOP) (LOADINV TOP) (OEN TOP) (OENINV TOP)))
                 (COMPLEMENT-OUT NIL) (DRIVING-CAP NO))
```

```
("reg2port" (PARAMETERS (N)) (AREA (* N 107 48))
            (DELAY 2)
            (ONE-BIT-DELAY 2)
            (RIPPLE-DELAY 0)
            (RIPPLE-OFFSET 0)
            (DATA-TERMINAL (OUT (IN)))
            (POWER-TERMINAL (Vdd GND))
            (CTL-IN-TERMINAL
             ((LOAD (AND LOAD CK1))
              (LOADINV (NOT (AND LOAD CK1)))
              (OEN (AND CK2 OEN))
              (OENINV (NOT (AND OEN CK2)))))
            (CTL-OUT-TERMINAL NIL)
            (CTL-TERM-EDGE
             ((LOAD TOP) (LOADINV TOP) (OEN TOP) (OENINV TOP)))
            (COMPLEMENT-OUT NIL) (DRIVING-CAP NO))
("scanreg" (PARAMETERS (N)) (AREA (* N 66 199)) (DELAY 2)
           (ONE-BIT-DELAY 2)
           (RIPPLE-DELAY 0)
           (RIPPLE-OFFSET 0)
           (DATA-TERMINAL (OUT (IN)))
           (POWER-TERMINAL (Vdd GND))
           (CTL-IN-TERMINAL
            (SCANIN (LOAD LOAD) (LOADINV (NOT LOAD))
                    (PHI1 CK1) (PHI1INV (NOT CK1))
                    (PHI2 CK2) (PHI2INV (NOT CK2))
                    (SCAN SCAN) (SCANINV SCANINV)
                    (KEEP KEEP) (KEEPINV KEEPINV)))
           (CTL-OUT-TERMINAL SCANOUT)
           (CTL-TERM-EDGE
            ((LOAD TOP) (LOADINV TOP) (PHI1 TOP) (PHI1INV TOP)
             (PHI2INV TOP) (SCAN TOP) (SCANINV TOP) (KEEP TOP)
             (KEEPINV TOP) (SCANIN TOP) (SCANOUT BOTTOM)
             (PHI2 TOP)))
           (COMPLEMENT-OUT MASTER) (DRIVING-CAP SMALL))
("scanregmx" (PARAMETERS (N)) (AREA (* N 66 199)) (DELAY 2)
             (ONE-BIT-DELAY 2)
             (RIPPLE-DELAY 0)
             (RIPPLE-OFFSET 0)
             (DATA-TERMINAL (OUT (IN)))
             (POWER-TERMINAL (Vdd GND))
             (CTL-IN-TERMINAL
              (SCANIN (LOAD LOAD) (LOADINV (NOT LOAD))
                      (PHI1 CK1) (PHI1INV (NOT CK1))
                      (PHI2 CK2) (PHI2INV (NOT CK2))
                      (SCAN SCAN) (SCANINV SCANINV)
                      (KEEP KEEP) (KEEPINV KEEPINV)))
             (CTL-OUT-TERMINAL SCANOUT)
             (CTL-TERM-EDGE
              ((LOAD TOP) (LOADINV TOP) (PHI1 TOP)
               (PHI1INV TOP) (PHI2INV TOP) (KEEP TOP)
               (KEEPINV TOP) (SCANOUT BOTTOM) (PHI2 TOP)
               (SCANIN TOP) (SCAN TOP) (SCANINV TOP)))
             (COMPLEMENT-OUT MASTER) (DRIVING-CAP SMALL))
```

```
     ("scanmslatch" (PARAMETERS (N)) (AREA (* N 52 108)) (DELAY 1)
                    (ONE-BIT-DELAY 1)
                    (RIPPLE-DELAY 0)
                    (RIPPLE-OFFSET 0)
                    (DATA-TERMINAL (OUT (IN)))
                    (POWER-TERMINAL (Vdd GND))
                    (CTL-IN-TERMINAL
                     (SCANIN (LOAD LOAD) (LOADINV (NOT LOAD))
                             (PHI1 CK1) (PHI1INV (NOT CK1))
                             (SHIFT SCAN) (SHIFTINV SCANINV)))
                    (CTL-OUT-TERMINAL SCANOUT)
                    (CTL-TERM-EDGE
                     ((LOAD TOP) (LOADINV TOP) (PHI1 TOP)
                      (PHI1INV TOP)
                      (SHIFT TOP) (SHIFTINV TOP) (SCANOUT BOTTOM)
                      (SCANIN TOP)))
                    (COMPLEMENT-OUT NIL) (DRIVING-CAP SMALL)))
(MUX ("mux2to1" (PARAMETERS (N)) (AREA (* N 52 69)) (DELAY 2)
                (ONE-BIT-DELAY 2)
                (RIPPLE-DELAY 0)
                (RIPPLE-OFFSET 0)
                (DATA-TERMINAL (OUT (IN1 IN2)))
                (POWER-TERMINAL (Vdd GND))
                (CTL-IN-TERMINAL ((SEL1 SELECT)
                                  (SEL2 (not SELECT))))
                (CTL-OUT-TERMINAL NIL)
                (CTL-TERM-EDGE ((SEL1 TOP) (SEL2 TOP)))
                (COMPLEMENT-OUT OUTINV)
                (DRIVING-CAP NO)))
(NOP ("adder" (PARAMETERS (N)) (AREA (* N 48 214))
              (DELAY (+ 6 (* N 2)))
              (ONE-BIT-DELAY 6)
              (RIPPLE-DELAY (* N 2))
              (RIPPLE-OFFSET 0)
              (DATA-TERMINAL (OUT (IN1)))
              (POWER-TERMINAL (Vdd GND))
              (CTL-IN-TERMINAL ((CIN GND) (CININV VDD)))
              (CTL-OUT-TERMINAL COUT)
              (CTL-TERM-EDGE ((CIN BOTTOM) (CININV BOTTOM)
              (COUT TOP)))
              (COMPLEMENT-OUT OUTINV) (DRIVING-CAP NO)))
(+ ("adder" (PARAMETERS (N)) (AREA (* N 48 214))
            (DELAY (+ 6 (* N 2)))
            (ONE-BIT-DELAY 6)
            (RIPPLE-DELAY (* N 2))
            (RIPPLE-OFFSET 0)
            (DATA-TERMINAL (OUT (IN1 IN2)))
            (POWER-TERMINAL (Vdd GND))
            (CTL-IN-TERMINAL ((CIN GND) (CININV VDD)))
            (CTL-OUT-TERMINAL COUT)
            (CTL-TERM-EDGE ((CIN BOTTOM) (CININV BOTTOM)
                            (COUT TOP)))
            (COMPLEMENT-OUT OUTINV) (DRIVING-CAP NO))
```

```
   ("add_sub" (PARAMETERS (N)) (AREA (* N 48 324))
              (DELAY (+ 9 (* N 2)))
              (ONE-BIT-DELAY 9)
              (RIPPLE-DELAY (* N 2))
              (RIPPLE-OFFSET 0)
              (DATA-TERMINAL (OUT (IN1 IN2)))
              (POWER-TERMINAL (Vdd GND))
              (CTL-IN-TERMINAL ((CIN GND) (CININV VDD)))
              (CTL-OUT-TERMINAL COUT)
              (CTL-TERM-EDGE ((CIN BOTTOM) (CININV BOTTOM)
              (COUT TOP)))
              (COMPLEMENT-OUT OUTINV) (DRIVING-CAP NO))
   ("ALU" (PARAMETERS (N)) (AREA (* N 100 500))
          (DELAY (+ 6 (* N 2))) (ONE-BIT-DELAY 6)
          (RIPPLE-DELAY (* N 2))
          (RIPPLE-OFFSET 0)
          (DATA-TERMINAL (OUT (IN1 IN2)))
          (POWER-TERMINAL (Vdd GND))
          (CTL-IN-TERMINAL
           ((CIN GND) (CININV VDD)
            (SELECT1 1) (SELECT2 0) (SELECT3 0)))
          (CTL-OUT-TERMINAL COUT)
          (CTL-TERM-EDGE ((CIN BOTTOM) (CININV BOTTOM) (COUT TOP)
                          (SELECT1 TOP) (SELECT2 TOP)
                          (SELECT3 TOP)))
          (COMPLEMENT-OUT OUTINV) (DRIVING-CAP NO)))
(- ("subtractor" (PARAMETERS (N)) (AREA (* N 48 245))
                 (DELAY (+ 7 (* N 2)))
                 (ONE-BIT-DELAY 7)
                 (RIPPLE-DELAY (* N 2))
                 (RIPPLE-OFFSET 0)
                 (DATA-TERMINAL (OUT (IN1 IN2)))
                 (POWER-TERMINAL (Vdd GND))
                 (CTL-IN-TERMINAL ((CIN Vdd) (CININV GND)))
                 (CTL-OUT-TERMINAL COUT)
                 (CTL-TERM-EDGE ((CIN BOTTOM) (CININV BOTTOM)
                                 (COUT TOP)))
                 (COMPLEMENT-OUT OUTINV) (DRIVING-CAP NO))
   ("add_sub" (PARAMETERS (N)) (AREA (* N 48 324))
              (DELAY (+ 9 (* N 2)))
              (ONE-BIT-DELAY 9)
              (RIPPLE-DELAY (* N 2))
              (RIPPLE-OFFSET 0)
              (DATA-TERMINAL (OUT (IN1 IN2)))
              (POWER-TERMINAL (Vdd GND))
              (CTL-IN-TERMINAL ((CIN VDD) (CININV GND)))
              (CTL-OUT-TERMINAL COUT)
              (CTL-TERM-EDGE ((CIN BOTTOM) (CININV BOTTOM)
                              (COUT TOP)))
              (COMPLEMENT-OUT OUTINV) (DRIVING-CAP NO))
```

```
   ("ALU" (PARAMETERS (N)) (AREA (* N 100 500))
          (DELAY (+ 7 (* N 2)))
          (ONE-BIT-DELAY 7)
          (RIPPLE-DELAY (* N 2))
          (RIPPLE-OFFSET 0)
          (DATA-TERMINAL (OUT (IN1 IN3)))
          (POWER-TERMINAL (Vdd GND))
          (CTL-IN-TERMINAL
           ((CIN Vdd) (CININV GND) (SELECT1 0) (SELECT2 1)
            (SELECT3 0)))
          (CTL-OUT-TERMINAL COUT)
          (CTL-TERM-EDGE ((CIN BOTTOM) (CININV BOTTOM) (COUT TOP)
                          (SELECT1 TOP) (SELECT2 TOP)
                          (SELECT3 TOP)))
          (COMPLEMENT-OUT OUTINV) (DRIVING-CAP NO)))
(NEGATE ("subtractor" (PARAMETERS (N)) (AREA (* N 48 245))
                      (DELAY (+ 7 (* N 2)))
                      (ONE-BIT-DELAY 7)
                      (RIPPLE-DELAY (* N 2))
                      (RIPPLE-OFFSET 0)
                      (DATA-TERMINAL (OUT (IN2)))
                      (POWER-TERMINAL (Vdd GND))
                      (CTL-IN-TERMINAL ((CIN Vdd) (CININV GND)))
                      (CTL-OUT-TERMINAL COUT)
                      (CTL-TERM-EDGE ((CIN BOTTOM) (CININV BOTTOM)
                                      (COUT TOP)))
                      (COMPLEMENT-OUT OUTINV) (DRIVING-CAP NO)))
(XOR ("xor2" (PARAMETERS (N)) (AREA (* N 40 50))
             (DELAY 1)
             (ONE-BIT-DELAY 1)
             (RIPPLE-DELAY 0)
             (RIPPLE-OFFSET 0)
             (DATA-TERMINAL (XOR2 (A B)))
             (POWER-TERMINAL (Vdd GND))
             (CTL-IN-TERMINAL NIL)
             (CTL-OUT-TERMINAL NIL)
             (CTL-TERM-EDGE NIL)
             (COMPLEMENT-OUT XOR2_INV) (DRIVING-CAP NO)))
(AND ("and2" (PARAMETERS (N)) (AREA (* N 40 50))
             (DELAY 1)
             (ONE-BIT-DELAY 1)
             (RIPPLE-DELAY 0)
             (RIPPLE-OFFSET 0)
             (DATA-TERMINAL (AND2 (A B)))
             (POWER-TERMINAL (Vdd GND))
             (CTL-IN-TERMINAL NIL)
             (CTL-OUT-TERMINAL NIL)
             (CTL-TERM-EDGE NIL)
             (COMPLEMENT-OUT AND2_INV) (DRIVING-CAP NO))
```

```
     ("ALU" (PARAMETERS (N)) (AREA (* N 100 500))
            (DELAY 1)
            (ONE-BIT-DELAY 1)
            (RIPPLE-DELAY 0)
            (RIPPLE-OFFSET 0)
            (DATA-TERMINAL (OUT2 (IN1 IN2)))
            (POWER-TERMINAL (Vdd GND))
            (CTL-IN-TERMINAL
             ((SELECT1 0) (SELECT2 0) (SELECT3 1)))
            (CTL-OUT-TERMINAL NIL)
            (CTL-TERM-EDGE
             ((SELECT1 TOP) (SELECT2 TOP) (SELECT3 TOP)))
            (COMPLEMENT-OUT NIL) (DRIVING-CAP NO)))
(OR ("or2" (PARAMETERS (N)) (AREA (* N 40 50))
           (DELAY 1)
           (ONE-BIT-DELAY 1)
           (RIPPLE-DELAY 0)
           (RIPPLE-OFFSET 0)
           (DATA-TERMINAL (OR2 (A B)))
           (POWER-TERMINAL (Vdd GND))
           (CTL-IN-TERMINAL NIL)
           (CTL-OUT-TERMINAL NIL)
           (CTL-TERM-EDGE NIL)
           (COMPLEMENT-OUT OR2_INV) (DRIVING-CAP NO)))
(NOT ("inverter" (PARAMETERS (N)) (AREA (* N 40 31))
                 (DELAY 1)
                 (ONE-BIT-DELAY 1)
                 (RIPPLE-DELAY 0)
                 (RIPPLE-OFFSET 0)
                 (DATA-TERMINAL (OUTINV (IN)))
                 (POWER-TERMINAL (Vdd GND))
                 (CTL-IN-TERMINAL NIL)
                 (CTL-OUT-TERMINAL NIL)
                 (CTL-TERM-EDGE NIL)
                 (COMPLEMENT-OUT NIL) (DRIVING-CAP NO)))
(! ("inverter" (PARAMETERS (N)) (AREA (* N 40 31))
               (DELAY 1)
               (ONE-BIT-DELAY 1)
               (RIPPLE-DELAY 0)
               (RIPPLE-OFFSET 0)
               (DATA-TERMINAL (OUTINV (IN)))
               (POWER-TERMINAL (Vdd GND))
               (CTL-IN-TERMINAL NIL)
               (CTL-OUT-TERMINAL NIL)
               (CTL-TERM-EDGE NIL)
               (COMPLEMENT-OUT NIL) (DRIVING-CAP NO)))
```

```
(DECODE ("constant" (PARAMETERS (N PATTERN))
                    (AREA (* N 13 48))
                    (DELAY 0)
                    (ONE-BIT-DELAY 0)
                    (RIPPLE-DELAY 0)
                    (RIPPLE-OFFSET 0)
                    (DATA-TERMINAL (OUT (IN)))
                    (POWER-TERMINAL (Vdd GND))
                    (CTL-IN-TERMINAL NIL)
                    (CTL-OUT-TERMINAL NIL)
                    (CTL-TERM-EDGE NIL)
                    (COMPLEMENT-OUT NIL) (DRIVING-CAP NO)))
(trist-BUF ("trist_buffer"  (PARAMETERS (N)) (AREA (* N 38 50))
                            (DELAY 1)
                            (ONE-BIT-DELAY 1)
                            (RIPPLE-DELAY 0)
                            (RIPPLE-OFFSET 0)
                            (DATA-TERMINAL (OUT (IN)))
                            (POWER-TERMINAL (Vdd GND))
                            (CTL-IN-TERMINAL
                             ((CNTL OEN) (CNTLINV (not OEN))))
                            (CTL-OUT-TERMINAL NIL)
                            (CTL-TERM-EDGE
                             ((CNTL TOP) (CNTLINV TOP)))
                            (COMPLEMENT-OUT NIL)
                            (DRIVING-CAP SMALL)))
(>> ("barrelR6L1" (PARAMETERS (N)) (RANGE (-6 1))
                  (AREA (* N 63 170)) (DELAY 4)
                  (ONE-BIT-DELAY 3)
                  (RIPPLE-DELAY 0)
                  (RIPPLE-OFFSET (- M))
                  (DATA-TERMINAL (OUT (IN)))
                  (POWER-TERMINAL (Vdd GND))
                  (CTL-IN-TERMINAL
                   ((SHFTL1 L1) (SHFT0 R0) (SHFT1 R1)
                    (SHFT2 R2) (SHFT3 R3) (SHFT4 R4)
                    (SHFT5 R5) (SHFT6 R6) (PRCHRG CK1)))
                  (CTL-OUT-TERMINAL NIL)
                  (CTL-TERM-EDGE
                   ((SHFTL1 TOP) (SHFT0 TOP) (SHFT1 TOP)
                    (SHFT3 TOP) (SHFT4 TOP) (SHFT5 TOP)
                    (SHFT2 TOP) (SHFT6 TOP) (PRCHRG TOP)))
                  (COMPLEMENT-OUT NIL) (DRIVING-CAP NO))
   ("shift" (PARAMETERS (N (SBY (STRUCTURE_RIGHT M))))
            (RANGE (-31 31))
            (AREA (+ (* (my-length SBY (list "up1" "down1")) 84)
                     (* (my-length SBY (list "up2" "down2")) 90)
                     (* (my-length SBY (list "up4" "down4")) 102)
                     (* (my-length SBY (list "up8" "down8")) 130)
                     (* (my-length SBY (list "up16" "down16")) 186)
                     ))
            (DELAY (* 1 (length SBY)))
```

```
            (ONE-BIT-DELAY (* 1 (length SBY)))
            (RIPPLE-DELAY 0)
            (RIPPLE-OFFSET (- M))
            (DATA-TERMINAL (OUT (IN)))
            (POWER-TERMINAL (Vdd GND))
            (CTL-IN-TERMINAL ((SHIFT BUS ((length SBY)
                                          (encoded_SHIFT SBY)))
                              (SHIFTBAR BUS
                                     ((length SBY)
                                      (NOT (encoded_SHIFT SBY))))
                              (MSBIN BUS
                                   ((length SBY) SHIFT_IN_NUM))))
            (CTL-OUT-TERMINAL NIL)
            (CTL-TERM-EDGE
             ((SHIFT TOP) (SHIFTBAR TOP) (MSBIN TOP)))
            (COMPLEMENT-OUT NIL) (DRIVING-CAP NO)))
(<< ("barrelR6L1" (PARAMETERS (N)) (RANGE (-1 6))
                  (AREA (* N 63 170))  (DELAY 4)
                  (ONE-BIT-DELAY 3)
                  (RIPPLE-DELAY 0)
                  (RIPPLE-OFFSET M)
                  (DATA-TERMINAL (OUT (IN)))
                  (POWER-TERMINAL (Vdd GND))
                  (CTL-IN-TERMINAL
                   ((SHFTL1 L1) (SHFT0 R0) (SHFT1 R1)
                    (SHFT2 R2) (SHFT3 R3) (SHFT4 R4)
                    (SHFT5 R5) (SHFT6 R6) (PRCHRG CK1)
                    (DIN SHIFT_IN_NUM)))
                  (CTL-OUT-TERMINAL NIL)
                  (CTL-TERM-EDGE
                   ((SHFTL1 TOP) (SHFT0 TOP) (SHFT1 TOP)
                    (SHFT3 TOP) (SHFT4 TOP) (SHFT5 TOP) (SHFT6 TOP)
                    (SHFT2 TOP) (PRCHRG TOP) (DIN BOTTOM)))
                  (COMPLEMENT-OUT NIL) (DRIVING-CAP NO))
   ("shift" (PARAMETERS (N (SBY (STRUCTURE_LEFT M))))
            (RANGE (-31 31))
            (AREA (* N 50
                     (+ (* (my-length SBY (list "up1" "down1")) 84)
                        (* (my-length SBY (list "up2" "down2")) 90)
                        (* (my-length SBY (list "up4" "down4")) 102)
                        (* (my-length SBY (list "up8" "down8")) 130)
                        (* (my-length SBY (list "up16" "down16"))
                           186)
                        )))
            (DELAY (* 1 (length SBY)))
            (ONE-BIT-DELAY (* 1 (length SBY)))
            (RIPPLE-DELAY 0)
            (RIPPLE-OFFSET M)
            (DATA-TERMINAL (OUT (IN)))
            (POWER-TERMINAL (Vdd GND))
```

```
                (CTL-IN-TERMINAL
                 ((SHIFT BUS ((length SBY)
                              (encoded_SHIFT SBY)))
                  (SHIFTBAR BUS ((length SBY)
                                 (NOT (encoded_SHIFT SBY))))
                  (LSBIN BUS ((length SBY) SHIFT_IN_NUM))))
                (CTL-OUT-TERMINAL NIL)
                (CTL-TERM-EDGE
                 ((SHIFT TOP) (SHIFTBAR TOP) (LSBIN BOTTOM)))
                (COMPLEMENT-OUT NIL) (DRIVING-CAP NO)))
(@ ("transfer" (PARAMETERS (N)) (AREA (* N 107 48)) (DELAY 0)
                (RIPPLE-DIR NIL)
                (ONE-BIT-DELAY 0)
                (RIPPLE-DELAY 0)
                (TIMING-CONSTRAINT NIL)
                (DATA-TERMINAL (OUT (IN1)))
                (POWER-TERMINAL (Vdd GND))
                (CTL-IN-TERMINAL NIL)
                (CTL-OUT-TERMINAL NIL)
                (CTL-TERM-EDGE NIl)
                (COMPLEMENT-OUT NIL) (DRIVING-CAP NO)))
("#" ("transfer" (PARAMETERS (N)) (AREA (* N 107 48)) (DELAY 0)
                  (RIPPLE-DIR NIL)
                  (ONE-BIT-DELAY 0)
                  (RIPPLE-DELAY 0)
                  (TIMING-CONSTRAINT NIL)
                  (DATA-TERMINAL (OUT (IN1)))
                  (POWER-TERMINAL (Vdd GND))
                  (CTL-IN-TERMINAL NIL)
                  (CTL-OUT-TERMINAL NIL)
                  (CTL-TERM-EDGE NIL)
                  (COMPLEMENT-OUT NIL) (DRIVING-CAP NO)))
(= ("transfer" (PARAMETERS (N)) (AREA 0) (DELAY 0)
                (RIPPLE-DIR NIL)
                (ONE-BIT-DELAY 0)
                (RIPPLE-DELAY 0)
                (TIMING-CONSTRAINT NIL)
                (DATA-TERMINAL (OUT (IN1)))
                (POWER-TERMINAL (Vdd GND))
                (CTL-IN-TERMINAL NIL)
                (CTL-OUT-TERMINAL NIL)
                (CTL-TERM-EDGE NIL)
                (COMPLEMENT-OUT NIL) (DRIVING-CAP NO)))
)
```

Keyword	Modules	Description
INWIDTH	RAM, ROM, FSM, PLA	input width in bit
OUTWIDTH	RAM, ROM, FSM, PLA	output width in bit
INPLANE	RAM, ROM, FSM, PLA	map of input plane
OUTPLANE	RAM, ROM, FSM, PLA	map of output plane
MINTERM	FSM, PLA	number of min-terms
BDSYN	FSM, PLA	bds file of the module
WIDTH	FIFO, reg file	word width
LENGTH	FIFO, reg file, counter, timer	length of the module
INIT	counter	initial value of counter
N	multiplier	output bit width
N1	multiplier	bit width of multiplicand
N2	multiplier	bit width of multiplier
USER_DEFINE	all	given by user in silicon compilation

Table B.8: Keyword in the Parameters Attribute of Array Database.

B.5 Database for Array Modules

The database format for the array modules is very similar to the database format for the data path modules except for some attributes. The attributes in the array database are listed as follows with the emphasis on the difference between the attributes of the data path database and those of the array database:

parameters Similar to the data path database except that a different set of keywords are defined for various modules. Since most hardware parameter names of array modules are lower-case, quotes are needed for the parameter names. The functions recognized by the hardware mapper in this attribute are listed in Table B.4 and the keywords are listed in Table B.8.

area Similar to the data path database except that the area of some modules cannot be decided until the logic synthesis is performed. Examples of such modules are PLA's and FSM's. For those modules, the area attribute just shows the proportion of the parameters to the area.

delay Similar to the data path database. Again, delay of some modules cannot be decided until the logic synthesis step. Therefore, the delay attribute only shows the proportion of the parameters to the delay.

one-bit-delay Same as the data path database.

ripple-delay Same as the data path database.

ripple-offset Same as the data path database.

data-terminal Similar to the data path database except that multiple-port input/output is supported for memory modules. The format for single-port (one input port and one output port) memory blocks is "(output-data-term (address-term input-data-term))", which is the same as the data-terminal attribute in the data path database[1]. The format for multiple-port memory is as follows:

```
([(p1 d1) (p2 d2) ...] [(p1 a1 d1) (p2 a2 d2) ...])
```

where p_i is a port name, a_i is the address terminal name of p_i and d_i is the input/output data terminal name of p_i. The first bracket contains output terminals and the second bracket contains input terminals. The orders of the elements in the brackets are not important; however, the orders in the parentheses are important. Again, the input data terminals can be omitted for ROM modules.

power-terminal Same as the data path database.

ctl-in-terminal Same as the data path database.

complement-out Same as the standard cell database.

driving-cap Same as the standard cell database.

reducibility This attribute specifies if the module is tri-state buffered (i.e. multiplexer reducible). The value is t if it is, nil otherwise.

The following subsection is a print-out of the current array database of HYPER. Notice that some of the modules in the Lager cell library do not have the required information and since the print-out is mainly for demonstrating the database format, the missing information will be given an appropriate expression based on the correct format. Also keep in mind that the database is read by the Lisp interpreter and hence there is no difference between the lower-case letters and the upper-case letters.

[1]The input data terminal can be omitted for a ROM module.

B.5.1 Array Module Database in HYPER

```
; Need to define keywords for different modules to interpret
; parameters. The interpretation can also be an expression,
; eg. (log OUTWIDTH 2)
; for ram, rom : INWIDTH OUTWIDTH INPLANE OUTPLANE
; for fifo : WIDTH LENGTH
; for fsm, pla : INWIDTH OUTWIDTH BDSYN (INPLANE OUTPLANE MINTERM)
; for reg file :  WIDTH, LENGTH
; for counter : LENGTH INIT ...
; for multiplier : N1, N2, N
; No CTL-OUT-TERMINAL needed. However, REDUCIBILITY is needed.

((* ("mult" (PARAMETERS (("n" N1) ("m" N2) ("s" USER_DEFINE)
                         ("csindex" USER_DEFINE)))
            (AREA (* (+ 375 (* 129 (- (max "n" "m") 1)))
                     (+ 342 (* 135 (+ 1 (min ("n" "m")))))))
            (DELAY (+ 30 2
                      (* 3 (+ 2 (/ (ceiling (min (- "n" 1)
                                                 (- "m" 3))) 2)))))
            (ONE-BIT-DELAY 2)
            (RIPPLE-DELAY (* 3 (+ 2 (/ (ceiling (min (- "n" 1)
                                                     (- "m" 3))) 2))))
            (RIPPLE-OFFSET 0)
            (DATA-TERMINAL (P (X Y)))
            (POWER-TERMINAL (Vdd GND))
            (CTL-IN-TERMINAL NIL)
            (COMPLEMENT-OUT NIL) (DRIVING-CAP NO)
            (REDUCIBILTY NIL)))
 (ram ("RAM3T" (PARAMETERS (("in-width" INWIDTH)
                            ("out-width" OUTWIDTH)
                            ("ram-address-plane" INPLANE)
                            ("ram-bit-plane" OUTPLANE)))
               (AREA (* "in-width" "out-width" 31 28))
               (DELAY (+ (* 2 "in-width") (* 2 "out-width")))
               (DATA-TERMINAL (RAMDATABUS (RAMADDRESS RAMDATABUS)))
               ; mul-port     ([(p1 d1) ..] [(p1 a1 d1) ...])
               ; 1-port       (d [a d])
               (POWER-TERMINAL (Vdd GND))
               (CTL-IN-TERMINAL ((PHI1 CK2) (PHI2 CK1)
                                 (READ OEN) (WRITE LOAD)))
               (COMPLEMENT-OUT OUTINV) (DRIVING-CAP SMALL)
               (REDUCIBILITY NIL))
      ("dpram" (PARAMETERS (("width" OUTWIDTH)
                            ("words" (expt 2 INWIDTH))))
               (AREA (* "width" "words" 31 28))
               (DELAY (* 2 "width"))
               (DATA-TERMINAL (((PORT1 IN) (PORT2 OUT))
                               ((PORT1 WRITE_ADDR IN)
                                (PORT2 READ_ADDR OUT))))
               (POWER-TERMINAL (Vdd GND))
               (CTL-IN-TERMINAL ((PRE (not CK1)) (WRITE LOAD)))
               (COMPLEMENT-OUT NIL) (DRIVING-CAP SMALL)
               (REDUCIBILITY NIL)))
```

```
(rom ("ROM" (PARAMETERS (("in-width" INWIDTH)
                         ("out-width" OUTWIDTH)
                         ("rom-address-plane" INPLANE)
                         ("rom-bit-plane" OUTPLANE)))
            (AREA (* "in-width" "out-width" 159 20))
            (DELAY (+ (* 2 "in-width") (* 2 "out-width")))
            (DATA-TERMINAL (OUT (ADDRESS)))
            ; mul-port ([(p1 d1) ..] [(p1 a1) ..])
            ; 1-port (d [a])
            (POWER-TERMINAL (Vdd GND))
            (CTL-IN-TERMINAL ((PHI1 CK2) (PHI2 CK1)))
            (COMPLEMENT-OUT OUTINV) (DRIVING-CAP SMALL)
            (REDUCIBILITY NIL)))
(fifo ("FIFO" (PARAMETERS (("width" WIDTH) ("length" LENGTH)))
              (AREA (* 59 73 "length" "width"))
              (DELAY (* 3 (sqrt N)))
              (DATA-TERMINAL (OUT (IN)))
              (POWER-TERMINAL (Vdd GND))
              (CTL-IN-TERMINAL ((PHI1 CK2) (PHI2 CK1)))
              (COMPLEMENT-OUT OUTINV) (DRIVING-CAP small)
              (REDUCIBILITY NIL)))
(pla ("pla" (PARAMETERS (("inwidth" INWIDTH) ("outwidth" OUTWIDTH)
                         ("minterm"  MINTERM)
                         ("input-plane" INPLANE)
                         ("output-plane" OUTPLANE)))
            (AREA (* "minterm" (+ "inwidth" "outwidth")))
            (DELAY 5)
            (DATA-TERMINAL (OUT (IN)))
            (POWER-TERMINAL (Vdd GND))
            (CTL-IN-TERMINAL ((CLOCK CK1)))
            (COMPLEMENT-OUT NIL) (DRIVING-CAP NIL)
            (REDUCIBILITY NIL)))
(fsm ("fsm_bdsyn" (PARAMETERS (("inwidth" INWIDTH)
                               ("outwidth" OUTWIDTH)
                               ("bdsyn" BDSYN)))
                  (AREA (+ "inwidth" "out-width"))
                  ; use the same leafcell as ROM
                  (DELAY 5)
                  (DATA-TERMINAL (OUT (IN)))
                  (POWER-TERMINAL (Vdd GND))
                  (CTL-IN-TERMINAL ((PHI1 CK2) (PHI2 CK1)))
                  (COMPLEMENT-OUT NIL) (DRIVING-CAP NIL)
                  (REDUCIBILITY NIL)))
(reg-file ("latch" (PARAMETERS (("width" WIDTH)))
                   (AREA (* "width" 20 164))
                   (DELAY 2)
                   (DATA-TERMINAL (OUT (IN)))
                   (POWER-TERMINAL (Vdd GND))
                   (CTL-IN-TERMINAL ((LD LOAD) (PHI1 CK1)
                                               (PHI2 CK2)))
                   (COMPLEMENT-OUT NIL) (DRIVING-CAP NIL)
                   (REDUCIBILITY NIL)))
```

```
(counter ("lpc" (PARAMETERS (("initcnt" INIT) "num_of_loopslice"
                             "detslice" "max_loop_size"))
                (AREA (* 50 263 (log "max_loop_size" 2)))
                (DELAY 5)
                (DATA-TERMINAL (LPCCOUNT NIL))
                (POWER-TERMINAL (Vdd GND))
                (CTL-IN-TERMINAL ((INC count) (ENA (not RESET))
                                  (PHI1 CK1) (PHI2 CK2)))
                (CTL-OUT-TERMINAL DETECT)
                (COMPLEMENT-OUT LPCCOUNTBAR) (DRIVING-CAP NIL)
                (REDUCIBILITY NIL))
         ("pc" (PARAMETERS (("width" LENGTH) ("initcnt" INIT)))
               (AREA (* "width" 50 263))
               (DELAY 5)
               (DATA-TERMINAL ((COUNT_L COUNT_R) NIL))
               (POWER-TERMINAL (Vdd GND))
               (CTL-IN-TERMINAL ((PHI1 CK1) (PHI2 CK2)
                                 (RESET1 RESET)
                                 (RESET2 0) (RESET3 0)))
               (CTL-OUT-TERMINAL COUT)
               (COMPLEMENT-OUT (COUNT_L_INV COUNT_R_INV))
               (DRIVING-CAP NIL)
               (REDUCIBILITY NIL)))
(timer ("timer" (PARAMETERS (("width" LENGTH)))
                (AREA (* "width" 50 263))
                (DELAY 5)
                (DATA-TERMINAL((TIMERCOUNT_L TIMERCOUNT_R)
                               (INITCOUNT)))
                (POWER-TERMINAL (Vdd GND))
                (CTL-IN-TERMINAL ((RESET1INV (and (not RESET)
                                                  CK2))
                                  (RESET2 (and CK2 RESET))))
                (CTL-OUT-TERMINAL COUT)
                (COMPLEMENT-OUT (TIMERCOUNT_L_INV
                                 TIMERCOUNT_R_INV))
                (DRIVING-CAP NIL)
                (REDUCIBILITY NIL)))
)
```

B.6 Conclusion

This appendix describes the format of the hardware database in the HYPER synthesis system. This format is in the Lisp syntax and forms a hierarchical database. Three databases, which contain information on data path modules, array modules, and standard cells respectively, have been built. All the databases have the same structure and the attributes to specify the area, delay, and wiring information. However, the format is somewhat different for various applications. These databases have been proved useful for many real applications. Nevertheless, more features such as testability and more elaborate delay information may be needed in the future to cope with various design issues.

Appendix C

Flow Graph Format For Hardware Mapper

C.1 Introduction

This document describes the HYPER flow graph format for the hardware mapping process. This format is a Lisp description of hierarchical control/data flow graphs. The author would assume that readers have the basic Common Lisp background in reading this documentation. The best way to understand the flow graph format is by going through a complete example. An example of this format can be found in Section 10 of this Chapter, which is a 7th order IIR filter. Other examples can be found in čhu/synthesis/graph2sdl/NEW/version-eps/eps1.dfg on the zabriskie cluster, which is the flow graph description of the epsilon processor, and in čhu/synthesis/graph2sdl/vtb/v4/vtb.dfg, which is the flow graph description of the Viterbi processor.

Five hierarchical constructs are supported in the flow graph format – if, for (iteration), waitfor, func, and process. The first three are control macros and the other two are hierarchical nodes to represent a hardware function or a complete Silage description. The usage of these constructs will be further discussed in the following sections. The mux statements in Silage are different from the if constructs described here. So are the loop statements in Silage different from the for constructs. The mux statements in Silage will be treated as mux nodes in the flow graph format and they are data nodes. Similarly, the loop statements in Silage will be fully expanded at this level and be treated as data nodes. On

the other hand, if statements and for statements in Silage are treated as control macros at this level and are associated with control states and subgraphs. Their executions depend on the input data as well as the states of the Finite State Machine (FSM).

At this level of abstraction, delay nodes in the Silage description have been properly deleted and replaced by assign nodes (i.e. equal nodes) and control constraints. Control constraints are handled by the HYPER scheduler and will be discarded by the hardware mapper. Some other primitive nodes in the HYPER ASCII flow graphs [28], such as cast nodes, will also be ignored at this level of abstraction.

A transformation process before the hardware mapper not only deletes some improper nodes for the hardware mapper, but also allocates some operations which are not described explicitly in a Silage program but are implied in the program. For example, inside the subgraph of a for node, the transformation step will put down an index increment node and some nodes for breaking the iteration. Another example can be described as follows: if the user writes the following Silage code:

```
a = a@1 + a@2;
```

The transformation program should be able to detect that this statement involves an implicit register transfer and therefore replace it by the following two statements:

```
b = a@1;
a = a@1 + b;
```

Notice that timing constraints should also be imposed so that the register transfer and the addition are performed at the same time. This can be illustrated using Figure C.1.

A flow graph description for the hardware mapper (called the decorated flow graph) can be divided into six parts – *node-list*, *edge-list*, parent-node, subgraph, in-edge/out-edge, and control-step. Each of these parts will be described in the following sections. Several short examples and notes will also be given in this appendix to illustrate the major ideas. Questions and comments of the flow graph format should be mailed to chu@zabriskie.berkeley.edu.

C.2 Definition of Graph

A graph contains a node-list and an edge-list. To facilitate the mapping process, an in-edge-list which consists of the input edges of the graph and an out-edge-list which

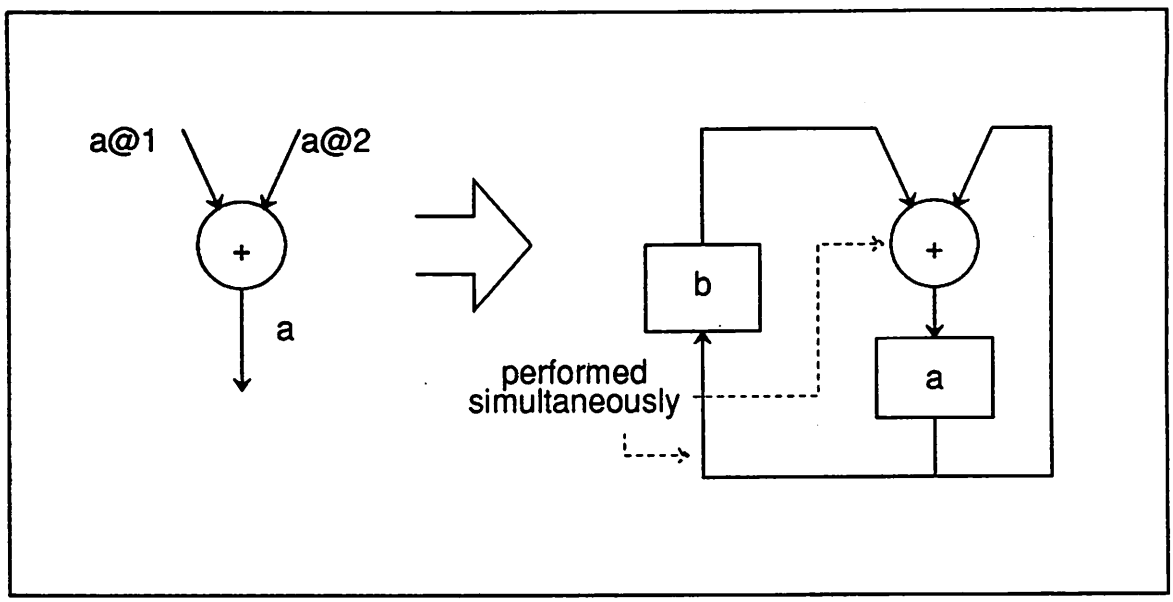

Figure C.1: Major Steps of Data Path Generation

consists of the output edges of the graph are also included in the graph description. The reason for this structure will become clearer later in the subgraph section of this report. The ordering of the nodes and the edges in the node-list and edge-list is not important except that the first node of the node-list has to be a process node which has a subgraph containing the highest hierarchy of the whole flow graph. Once the nodes and the edges have been described in the node-list and the edge-list, the order of the nodes and the edges in the lists will be used to assign the in/out-edge information of a node and the in/out-node information of an edge.

A node in the flow graph format contains the following information:

```
node-definition :=
    (defstruct node
              (name nil)
              (width 0)
              (in-edge nil)
              (out-edge nil)
              (function nil)
```

```
            (implement nil)
            (subgraph nil)
            (ctl-constraint nil)
            (attribute nil)
    )
```

The default value of each attribute is given as the second element of each parenthesis. Similarly, the definition of an edge can be written as follows:

```
edge-definition :=
    (defstruct edge
            (value nil)
            (width 1)
            (type 'data)
            (in-node nil)
            (out-node nil)
            (attribute nil)
    )
```

The detail usage of each item will be described in the following sections. Bear in mind that the flow graph description is a Common Lisp program which should be able to be compiled and loaded in a Common Lisp environment.

The flow graph description uses only Common Lisp functions with one exception. A function called "my-nth" which accepts two lists as parameters and returns a list is defined. My-nth works similar to the Lisp function "nth" except that the first argument is a number list instead of a number. A list of elements of the second argument is returned. The definition and an example of my-nth are as follows:

```
(defun my-nth (number-list a-list)
  (let ((return-list nil))
       (dolist (i number-list)
               (if (null i)
                   (setq return-list (cons 'nil return-list))
                   (setq return-list (cons (nth i a-list)
                                           return-list))))
```

```
        (setq return-list (reverse return-list))))

(my-nth '(1 2 4) '(a b c d e f)) = (b c e)
```

C.3 Node-list Description

This section describes how to specify the node-list in a flow graph description. A global variable called *node-list* is assigned to contain all the flow graph nodes. Each node is then given proper information including node-name, node-width, node-function, node-implement and node-attribute. Not all the information is needed for every node. If an attribute is not assigned a value, the default value of the attribute is assumed.

C.3.1 Node-name

Node-name of a node is used to designate the execution unit of the node in the decorated flow graph description. For example, the node-name of an addition node may be 'adder1 and the node-name of a read node may be 'RAM1. For nodes that do not need an execution unit such as control macro nodes, merge/field (bitmerge/bitselect) nodes, and assign nodes, their node names should not be given. Furthermore, for operations that are performed in the control path (control slices or finite state machines), their node-name attribute should not be specified either.

C.3.2 Node-implement

The node-implement field of a node specifies the implementation of the execution unit of the node. The data type of this field should be a character string to match the data type of the cell names in the hardware cell library. For more information about the hardware database library, please see the documentation in Appendix B. If the node-name of a node is not given, node-implement of this node should not be given either. In HYPER, node-implement is decided by the hardware selection process. User can also assign the implementation by using Silage pragma statements.[1]

[1]This feature has not been implemented in the current HYPER synthesis system yet.

C.3.3 Node-width

The node-width field specifies the width in bit of a flow graph node. For control macro nodes, func nodes, and process nodes, no value is needed. For merge/field nodes and read/write nodes which involve different widths of input edges and output edges, additional information related to the node widths and the edge widths may be required in the node-attribute field of this node. If an execution unit is allocated without specifying node-width, the hardware mapper will try to find out the width of the execution unit from its input/output edges.

C.3.4 Node-function

The node-function field specifies the function of a node. Table C.1 lists all the functions that are recognized by the hardware mapping program. These functions include five categories – control functions, I/O functions, assignment, merge/field, and functions supported in the hardware database library.

C.3.5 Node-attribute

Several functions need to have appropriate attributes to describe the required parameters. These attributes are organized as Common Lisp associate lists and are created by the *pairlis* function of Lisp in the flow graph description. In addition to the function names, Table C.1 also lists the attribute names needed for each function. The data types of these attributes are described in Table C.2.

C.4 Edge-list Description

A global variable *edge-list* is specified to contain all the edges (in all hierarchies) in the given flow graph. This is done in the following fashion:

```
(setq *edge-list*
      (list
       (make-edge :value edge-value1 ;; other attributes)
       (make-edge :value edge-value2 ;; other attributes)
       ;; other edges
       ))
```

Description	Function	Inputs	Outputs	Attributes
process	process	(*)	(*)	none
func	func	(*)	(*)	none
waitfor	waitfor	(Cond *)	(*)	none
if	if	(Cond *)	(*)	none
for(iteration)	for	(*)	(*)	none
merge	merge	(*)	(Out)	none
field	field	(In)	(Out)	(field)
read	read	(Adr)	(Out)	(storage port parval)
write	write	(In Adr)	None	(storage port parval)
assign	=	(In)	(Out)	none
add	+	(In1 In2)	(Out)	none
subtract	−	(In1 In2)	(Out)	none
inc	++	(In)	(Out)	none
cond_ge	>=	(In1 In2)	(Out)	none
cond_e	==	(In1 In2)	(Out)	none
cond_le	<=	(In1 In2)	(Out)	none
mux	mux	(In1 In2 Cond)	(Out)	none
trist_buf	trist-buf[1]			
register	reg[1]			
const_decode	decode[1]			
and	and	(In1 In2)	(Out)	none
or	or	(In1 In2)	(Out)	none
not	not	(In)	(Out)	none
right_shift	>>	.(In)	(Out)	(shift shift-in-num)
left_shift	<<	(In)	(Out)	(shift shift-in-num)

Table C.1: Node Function Handled by Hardware Mapper.

* Variable number of inputs/outputs

[1] These functions exist in the hardware database, but they are not used for node functions.

attribute name	data type	description	example
field	list	(Index1 Index2)	(32 18)
storage	symbol	storage type	'ROM*
port	symbol	port name	'port1
parval	list	lists of parval	(("width" 32) ("length" 10))[1]
shift	integer	No. of bits shifted	4
shift-in-num	0/1/sign	No. shifted in	'sign

Table C.2: Node Attribute.

* Should be a function in the array database.

[1] A list of (parameter-name parameter-value) pairs. If a value is not given, the hardware mapper will try to find the default value.

Each edge is a Common Lisp structure containing the following information: value, width, type, in-node, out-node and attribute. Similar to the node structure, not all the information is needed for every edge. If certain information is not given, default values are assumed.

C.4.1 Edge-value

Edge-value specifies the value of an edge. It can be a constant or a variable name (with proper index if necessary). Edge value doesn't need to be unique. Edges with the same constant value will be allocated as the same constant block if their widths are equal. Notice that edges with the same value and different widths should *not* be combined as one edge since they will be allocated as different constant blocks in the data path.

The same edges at different hierarchies should have the same edge-value with proper index modification. For example, an input edge of a for node may have an edge-value a[0] as the initial value of the loop. Inside the subgraph of the for node, this edge may be called a[i-1] to represent the value of the previous iteration. This correspondence is established by the ordering of the in-edge of the for node and the ordering of the input edge list of the subgraph of the for node. This will be further discussed in the following sections.

C.4.2 Edge-width

Edge-width gives the width of an edge. If the value is not specified, the hardware mapping program will try to decide the width by the widths of the edge's input/output nodes. If this doesn't work, the program will use some rules to decide the width of the edge. It is always advisory to specify the edge-width at the flow graph level since the rules used by the hardware mapper may lead to inconsistent results. These rules are listed below:
(1) For control edges (edge type is 'control), edge-width is 1.
(2) Derive edge-width by its input/output nodes.
(3) A constant edge may get edge-width equal to $\lceil log(value) \rceil$ if none of above rules work.
(4) Take the default value, i.e. 1.

C.4.3 Edge-type

Edge-type gives the type of an edge. Three types of edges are accepted by the hardware mapper – 'data, 'control, and 'break. A control edge always has edge-width 1. It is used as the conditions of multiplexers or for the control macro nodes. Control edges will be handled in the control path; therefore, no hardware allocation is needed. Break edges are special control edges of which the edge values are the conditions for breaking for loops. Data edges, unlike the control edges or the break edges, need to be properly allocated a storage unit such as a register or will otherwise be treated as a variable. The storage information of the data edges will be further discussed in the edge-attribute section.

C.4.4 Edge-in-node

Edge-in-node specifies the input nodes of an edge. It may be a node, nil, or a keyword 'parent. If edge-in-node is a node, the Common Lisp function "nth" is used to designate the node in the *node-list* to be the input node of the edge. When a parameter is passed from the *outside world* (i.e. outside of the subgraph), the edge-in-node attribute of the corresponding edge will be 'parent. That is, the edge-in-node attributes of all the input edges of a subgraph have the attribute value 'parent, meaning that the inputs of these edges are from outside of the subgraph. When an edge is not driven by any source node including the outside world, the value of its edge-in-node is nil. Examples of such edges are the constant edges.

attribute name	description	examples
storage	type of storage	'var, 'reg*
allocation	storage unit	'reg1, 'ROM1
implement	implementation	"reg2port"[1]

Table C.3: Edge Attribute.

* Should be a function in the hardware database or the keyword 'var.

[1] Should be a module name in the hardware database.

C.4.5 Edge-out-node

Edge-out-node of an edge is a list of keywords and/or nodes driven by the edge. It may also be an empty list. In the latter case, this edge doesn't have any fanout. Unlike node-in-edge or node-out-edge which will be discussed later, the order of the elements in the edge-out-node list is not important. The Lisp function "my-nth" is used to assign nodes in the *node-list* to be in the edge-out-node list. The usage of the keyword 'parent is similar to the usage of 'parent in the edge-in-node attribute. That is, whenever a value is going to passed out of the flow graph, it should have 'parent in its edge-out-node list. The keyword 'break may be used in the subgraph of a *for* node as the edge-out-node value of the break edge. When the edge-out-node of an edge is 'break, this edge is the condition of breaking the loop.

Edge-in-node and edge-out-node are duals of node-out-edge and node-in-edge. They can be derived from each other. We decided to keep both sets of information to facilitate the flow graph processing.

C.4.6 Edge-attribute

Edge-attribute contains all the information that is not included in the fields described above. It is organized as a Lisp associate list and the Lisp function "pairlis" is used to generate the list. Table C.3 lists several attributes which are necessary in a flow graph description.

For control edges and edges with constant values, attributes listed in Table C.3 are not required. For data edges, if the storage attribute of an edge is 'var, no allocation

or implement attribute is needed. Otherwise, all the three attributes are required. These attributes are decided by the hardware selection, assignment, and allocation processes, and should be parts of a decorated flow graph.

C.5 Parent-node Description of Edges

This section of the flow graph description specifies the 'parent attributes of edges. Each edge has a parent node which contains the edge. For example, edges of the highest hierarchy should have the process node as their parent node. This hierarchy information is specified in the parent-node description of edges.

The parent attributes can also be derived from the subgraph description which will be discussed in the next section. But since the hardware mapping process does not perform the derivation currently, this section of the flow graph description is necessary. To specify the 'parent attribute, the Lisp function acons is used. The following example shows how the assignment of the 'parent attribute of the first edge in the *edge-list* is performed: (assuming that node 0, i.e. the process node, is the parent node of the edge)

```
(setf (edge-attribute (nth 1 *edge-list*))
  (acons 'parent (nth 0 *node-list*) (edge-attribute (nth 1 *edge-list*))))
```

Similar assignments should be given for all the edges in the *edge-list*.

C.6 Subgraph Description of Nodes

Four types of nodes should have a subgraph description – process/func nodes, if nodes, for/iteration nodes, and waitfor nodes. For all the other nodes, this field should be given nil (the default value). A subgraph (or flow graph) structure is a list containing four lists – a node list, an edge list, an input-edge list (which is a sublist of the edge list), and an output-edge list (which is also a sublist of the edge list). The order of the nodes in the node list and the order of the edges in the edge list are not important; however, the order of the edges in the input edge list and the output edge list should match respectively to the order of the edges in the in-edge and out-edge lists of the node.

For a process node or a func node, the subgraph description is simply a flow graph description. An example is given as follows:

```
(setf (node-subgraph (nth 0 *node-list*))
  (list (my-nth '(1 2 3) *node-list*) ;; node list
        (my-nth '(0 1 2 3 14 15 22 23) *edge-list*) ;; edge list
        (my-nth '(0 14 15) *edge-list*) ;; input edge list
        (my-nth '(22 23) *edge-list*))) ;; output edge list
```

For a waitfor node, node-subgraph is a list consisting of two elements. The first element is the signal that the node is waiting for. It should be the edge-value of a control edge which is an input edge of the waitfor node. The second element of the node-subgraph is a flow graph description. For example, node one of *node-list* is a waitfor node and its subgraph is described as follows:

```
(setf (node-subgraph (nth 1 *node-list*))
  (list 'reset ;; conditions to be waited for
        (list (my-nth '(7 8 9 10 11) *node-list*) ;; node list
              (my-nth '(26 28 31 32 33 34) *edge-list*) ;; edge list
              (my-nth '(26) *edge-list*) ;; input edge list
              (my-nth '(31 32 33 34) *edge-list*)))) ;; output edge list
```

For a for/iteration node, the subgraph description is an associate list containing the following information: index, min, max, avg, ec, and subgraph. Index is the name of the loop index. Min, max, and avg are the minimum index value, the maximum index value and the average index value respectively. Ec is the termination condition of the for node and it can either be nil or the edge-value of a control edge inside the subgraph of the for node. Subgraph is a flow graph description. For example, node 2 of *node-list* is a for node and its subgraph description is given as follows:

```
(setf (node-subgraph (nth 2 *node-list*))
  (pairlis (list 'index 'min 'max 'avg 'ec 'subgraph)
           (list 'i      1    5000 1000 nil
                 (list (my-nth '(12 13 14) *node-list*) ;; node list
                       (my-nth '(37 38 39 40) *edge-list*) ;; edge list
                       (my-nth '(37) *edge-list*) ;; input edge list
                       (my-nth '(39 40) *edge-list*) ;; output edge list
                 ))))
```

Notice that the Lisp function "pairlis" is used in this description to generate an associate list for the node-subgraph description. Pairlis generates an associate list by pairing the corresponding elements of two lists.

For an if node, its subgraph is a list containing 2 or more elements. Each element is a list consisting of a condition and a subgraph to be executed under the condition. That is, the format of the subgraph of an if node is:

```
((condition1 subgraph1) (condition2 subgraph2) ...)
```

The keyword 'otherwise can be used as the last condition in the above list meaning that the subgraph following 'otherwise will be executed if none of the previous condition is true. For example, assuming that node 28 of *node-list* is an if node, its subgraph is given as follows:

```
(setf (node-subgraph (nth 28 *node-list*))
  (list (list 'eow ;; condition of performing subgraph1.
                   ;; It should be the edge-value of a control
                   ;; edge, which is one of the input edges of node 28.
              (list (my-nth '(29) *node-list*)
                    (my-nth '(59 60 61) *edge-list*)
                    (append (list nil) (my-nth (list 59 60) *edge-list*))
                    (my-nth '(61) *edge-list*)))
        (list 'otherwise
              (list (my-nth '(31) *node-list*)
                    (my-nth '(57 58) *edge-list*)
                    (append (my-nth (list 57) *edge-list*) (list nil nil))
                    (my-nth '(58) *edge-list*)))))
```

Notice that some nil's are given in the input edge lists of the subgraphs. This is to match the order of the input edges of the if node and the orders of the input edge lists. Some of the input edges of the if node are not needed in some of the subgraphs; therefore, nil's are given to fill out the spaces.

C.7 In/out-edge Description of Nodes

This section of the flow graph specifies the in-edge and out-edge fields of a node. Since the *edge-list* section comes after the *node-list* section in a flow graph description, the in-edge and out-edge information of a node can not be specified when the *node-list* is assigned. In-edge of a node is an ordered list of edges. The order of the list should match with the proper terminals of a cell in the hardware database which implements the node. For a control macro node or a func node, the order of the in-edges should match with the order of the input edges of the subgraph of the node. This order is very important because the hardware mapping procedure uses this information to trace flow graphs of different hierarchies.

Out-edge of a node is also an ordered list. For most cases, the out-edge list will be a single-element list. But for a control macro node, a func node, or a process node, the out-edge list will be a list of several edges. Again, the order of the out-edge list should match with the order of the output edges of the subgraph of the node. The following examples show how the in-edge and out-edge are assigned to node 1 of the *node-list*. In this example, the in-edge of node 1 is the first edge of the *edge-list* (edge number starts from 0) and the out-edge of node 1 is a list consisting of edge 1, 2, and 3 of the *edge-list*.

```
(setf (node-in-edge (nth 1 *node-list*))
      (my-nth '(0) *edge-list*))

(setf (node-out-edge (nth 1 *node-list*))
      (my-nth '(1 2 3) *edge-list*))
```

C.8 Control-step Description of Nodes

In this section of a decorated flow graph description, the control-steps of nodes are given. If a node is not to be executed by an execution unit, nor is it a hierarchical node, its control-step should be given nil. Examples of such nodes are merge nodes and field nodes. For each subgraph, the control-steps of the nodes in the subgraph start from 0. Hierarchies do not affect this property. However, the meaning of control steps for the lowest hierarchy of the flow graph is different from that for the other hierarchies of the flow graph. In the lowest hierarchy, control steps are the relative *clock cycles* to execute the operations. In the

higher hierarchies, on the other hand, control steps are the relative *order* of performing the operations.

The following statement specifies that node 1 of *node-list* is performed at control-step 0.

```
(setf (node-attribute (nth 1 *node-list*))
      (acons 'order 0 (node-attribute (nth 1 *node-list*))))
```

From this example, we can see that the control-step of a node is one of the attributes of the node with the attribute name 'order. Since the attribute of a node is a Common Lisp associate list, the Lisp function "acons" is used in this assignment.

C.9 Preprocessing

To interface with the rest of the HYPER synthesis system, the hardware mapper has incorporated a preprocessing phase to transform the format generated by Flow2Lisp (the Lisp flow graph generator from the ASCII flow graph) into the format described above. There are several tasks in the preprocessing phase. The most important ones are the management of transfer units and the transformation of broadcasting edges.

Transfer units are dummy units generated by the scheduling and allocation process to handle register transfer operations. Register transfer operations are in fact assign operations with both the input and the output edges decorated as registers. The preprocessing routine removes the decoration of the assign (or nop) nodes so that no transfer units will be generated in the hardware.

An edge can be decorated with multiple registers to represent the broadcasting of a calculated value to the registers. The following description shows an broadcasting example in which the value from node 1 is stored in both registerA and registerB[2].

```
(make-edge :value 'n29_state :width 32 :type 'data
           :attribute (pairlis '(storage allocation implement)
                               '(reg (registerA registerB) "reg2port"))
           :in-node (nth 1  *node-list*)
           :out-node (my-nth '(2 3) *node-list*))
```

[2]These two registers are the inputs to node 2 and node 3 respectively.

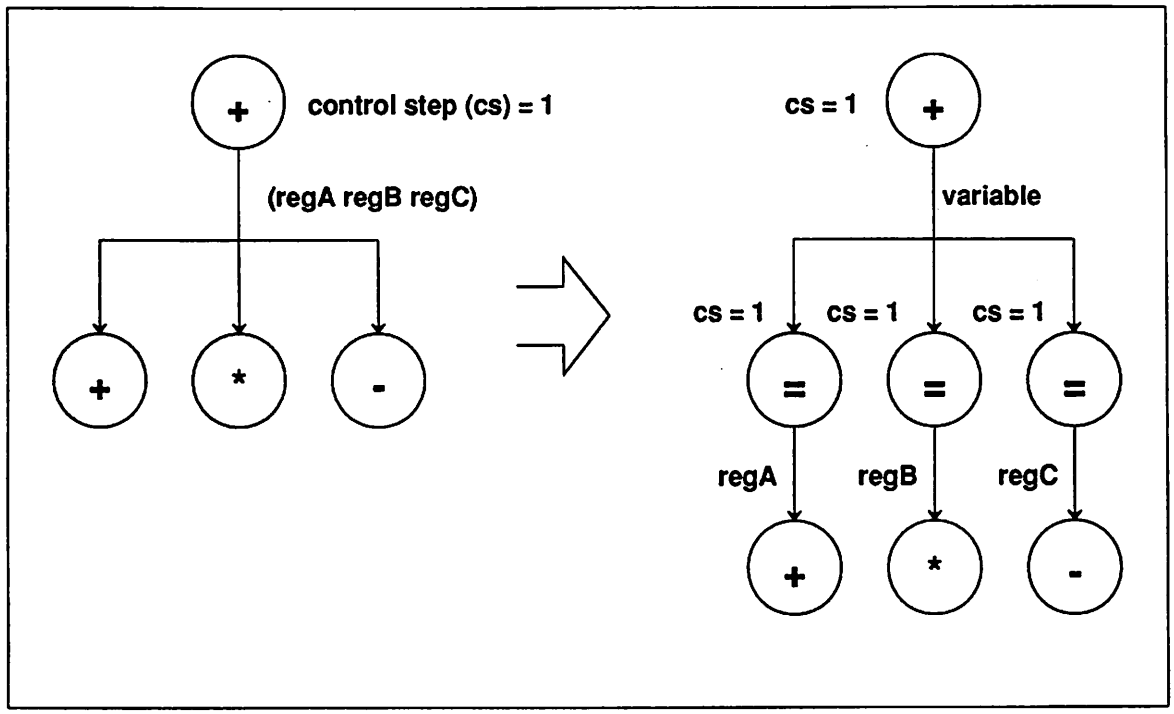

Figure C.2: Transformation to Handle Broadcasting

To handle various broadcasting cases, the flow graph format has been extended. The elements in the allocation list of an edge can be a register or nil (representing a variable). Similarly, the elements in the out-node list of an edge can now be a node, 'parent, or nil. The only constraint in the format is that the length of the allocation list should be equal to the length of the out-node list. The correspondence between the allocation list and the out-node list should be built to guarantee the correctness of the hardware generated. The only exception to this constraint is when an edge doesn't have any out-node. In this case, the out-node list can be nil even though the edge is decorated with multiple registers. The correspondence described above doesn't have to be true for this case either.

The preprocessing program performs a transformation to handle the edges with multiple-register decoration. This transformation can be illustrated using Figure C.2. The graph after the transformation becomes much cleaner and easier to process for the hardware mapper.

C.10 Flow Graph Example – IIR Filter

```
(setq *node-list*
      (list
       (make-node :function 'process)
       (make-node :function '+
                  :name 'a1
                  :implement "adder")
       (make-node :function '>>
                  :name 'b1
                  :implement "barrelR6L1"
                  :attribute (pairlis '(shift) '(2)))
       (make-node :function '+
                  :name 'a1
                  :implement "adder")
       (make-node :function '>>
                  :name 'b1
                  :implement "barrelR6L1"
                  :attribute (pairlis '(shift) '(2)))
       (make-node :function '+
                  :name 'a1
                  :implement "adder")
       (make-node :function '-
                  :name 's1
                  :implement "subtractor")
       (make-node :function '>>
                  :name 'b1
                  :implement "barrelR6L1"
                  :attribute (pairlis '(shift) '(2)))
       (make-node :function '+
                  :name 'a1
                  :implement "adder")
       (make-node :function '>>
                  :name 'b1
                  :implement "barrelR6L1"
                  :attribute (pairlis '(shift) '(1)))
       (make-node :function '=)
       (make-node :function '+
                  :name 'a1
                  :implement "adder")
       (make-node :function '+
                  :name 'a1
                  :implement "adder")
       (make-node :function '>>
                  :name 'b2
                  :implement "barrelR6L1"
                  :attribute (pairlis '(shift) '(2)))
       (make-node :function '+
                  :name 'a2
                  :implement "adder")
       (make-node :function '+
                  :name 'a2
                  :implement "adder")
```

```
(make-node :function '>>
           :name 'b2
           :implement "barrelR6L1"
           :attribute (pairlis '(shift) '(2)))
(make-node :function '-
           :name 's2
           :implement "subtractor")
(make-node :function '+
           :name 'a2
           :implement "adder")
(make-node :function '=)
(make-node :function '>>
           :name 'b2
           :implement "barrelR6L1"
           :attribute (pairlis '(shift) '(4)))
(make-node :function '+
           :name 'a2
           :implement "adder")
(make-node :function '+
           :name 'a2
           :implement "adder")
(make-node :function '>>
           :name 'b3
           :implement "barrelR6L1"
           :attribute (pairlis '(shift) '(2)))
(make-node :function '+
           :name 'a3
           :implement "adder")
(make-node :function '>>
           :name 'b3
           :implement "barrelR6L1"
           :attribute (pairlis '(shift) '(4)))
(make-node :function '-
           :name 's3
           :implement "subtractor")
(make-node :function '>>
           :name 'b3
           :implement "barrelR6L1"
           :attribute (pairlis '(shift) '(3)))
(make-node :function '+
           :name 'a3
           :implement "adder")
(make-node :function '+
           :name 'a3
           :implement "adder")
(make-node :function '+
           :name 'a3
           :implement "adder")
(make-node :function '=)
```

```
      (make-node :function '>>
                 :name 'b3
                 :implement "barrelR6L1"
                 :attribute (pairlis '(shift) '(2)))
      (make-node :function '-
                 :name 's3
                 :implement "subtractor")
      (make-node :function '+
                 :name 'a3
                 :implement "adder")
      (make-node :function '>>
                 :name 'b4
                 :implement "barrelR6L1"
                 :attribute (pairlis '(shift) '(3)))
      (make-node :function '+
                 :name 'a4
                 :implement "adder")
      (make-node :function '>>
                 :name 'b4
                 :implement "barrelR6L1"
                 :attribute (pairlis '(shift) '(2)))
      (make-node :function '-
                 :name 's4
                 :implement "subtractor")
      (make-node :function '+
                 :name 'a4
                 :implement "adder")
      (make-node :function '=)
      (make-node :function '+
                 :name 'a4
                 :implement "adder")
      (make-node :function '>>
                 :name 'b1
                 :implement "barrelR6L1"
                 :attribute (pairlis '(shift) '(6)))
      ))

(dolist (i *node-list*)
  (setf (node-width i) 32))

;;;;;;;;;;;;;;;;;;;;;; edge list ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setq *edge-list*
      (list
       (make-edge :value 'in
                  :type 'data
                  :in-node  'parent
                  :out-node (my-nth '(42) *node-list*)
                  :attribute (pairlis '(storage)
                                      '(var)))
```

```
(make-edge :value 'e1
           :type 'data
           :in-node (nth 42 *node-list*)
           :out-node (my-nth '(1) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regI1 "reg2port")))
(make-edge :value 'a
           :type 'data
           :in-node (nth 1 *node-list*)
           :out-node (my-nth '(12) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg rega1 "reg2port")))
(make-edge :value 'a[n-1]
           :type 'data
           :in-node nil
           :out-node (my-nth '(2 3 5 10 11) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg rega1 "reg2port")))
(make-edge :value 'a[n-2]
           :type 'data
           :in-node nil
           :out-node (my-nth '(11 7 8) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg rega2 "reg2port")))
(make-edge :value 'e5
           :type 'data
           :in-node (nth 10 *node-list*)
           :out-node nil
           :attribute (pairlis '(storage allocation implement)
                               '(reg rega2 "reg2port")))
(make-edge :value 'e6
           :type 'data
           :in-node (nth 2 *node-list*)
           :out-node (my-nth '(3) *node-list*)
           :attribute (pairlis '(storage)
                               '(var)))
(make-edge :value 'e7
           :type 'data
           :in-node (nth 3 *node-list*)
           :out-node (my-nth '(4) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg rega3 "reg2port")))
(make-edge :value 'e8
           :type 'data
           :in-node (nth 4 *node-list*)
           :out-node (my-nth '(5) *node-list*)
           :attribute (pairlis '(storage)
                               '(var)))
```

```
(make-edge :value 'e9
           :type 'data
           :in-node (nth 5 *node-list*)
           :out-node (my-nth '(6) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg rega3 "reg2port")))
;; 10
(make-edge :value 'e10
           :type 'data
           :in-node (nth 6 *node-list*)
           :out-node (my-nth '(1) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg rega3 "reg2port")))
(make-edge :value 'e11
           :type 'data
           :in-node (nth 7 *node-list*)
           :out-node (my-nth '(8) *node-list*)
           :attribute (pairlis '(storage)
                               '(var)))
(make-edge :value 'e12
           :type 'data
           :in-node (nth 8 *node-list*)
           :out-node (my-nth '(9) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg rega4 "reg2port")))
(make-edge :value 'e13
           :type 'data
           :in-node (nth 9 *node-list*)
           :out-node (my-nth '(6) *node-list*)
           :attribute (pairlis '(storage)
                               '(var)))
(make-edge :value 'e14
           :type 'data
           :in-node (nth 11 *node-list*)
           :out-node (my-nth '(12) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg rega4 "reg2port")))
(make-edge :value 'IN2
           :type 'data
           :in-node (nth 12 *node-list*)
           :out-node (my-nth '(18) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regI2 "reg2port")))
(make-edge :value 'b[n-1]
           :type 'data
           :in-node nil
           :out-node (my-nth '(13 14 19 20) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regb1 "reg2port")))
```

```
(make-edge :value 'b[n-2]
           :type 'data
           :in-node nil
           :out-node (my-nth '(16 17 21) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regb2 "reg2port")))
(make-edge :value 'e18
           :type 'data
           :in-node (nth 19 *node-list*)
           :out-node nil
           :attribute (pairlis '(storage allocation implement)
                               '(reg regb2 "reg2port")))
(make-edge :value 'e19
           :type 'data
           :in-node (nth 13 *node-list*)
           :out-node (my-nth '(14) *node-list*)
           :attribute (pairlis '(storage)
                               '(var)))
;; 20
(make-edge :value 'e20
           :type 'data
           :in-node (nth 14 *node-list*)
           :out-node (my-nth '(15) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regb3 "reg2port")))
(make-edge :value 'e21
           :type 'data
           :in-node (nth 16 *node-list*)
           :out-node (my-nth '(17) *node-list*)
           :attribute (pairlis '(storage)
                               '(var)))
(make-edge :value 'e22
           :type 'data
           :in-node (nth 17 *node-list*)
           :out-node (my-nth '(15) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regb4 "reg2port")))
(make-edge :value 'e23
           :type 'data
           :in-node (nth 15 *node-list*)
           :out-node (my-nth '(18) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regb4 "reg2port")))
(make-edge :value 'b
           :type 'data
           :in-node (nth 18 *node-list*)
           :out-node (my-nth '(22) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regb1 "reg2port")))
```

```
(make-edge :value 'e25
           :type 'data
           :in-node (nth 20 *node-list*)
           :out-node (my-nth '(21) *node-list*)
           :attribute (pairlis '(storage)
                               '(var)))
(make-edge :value 'e26
           :type 'data
           :in-node (nth 21 *node-list*)
           :out-node (my-nth '(22) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regb3 "reg2port")))
(make-edge :value 'IN3
           :type 'data
           :in-node (nth 22 *node-list*)
           :out-node (my-nth '(30) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regI3 "reg2port")))
(make-edge :value 'c[n-1]
           :type 'data
           :in-node nil
           :out-node (my-nth '(27 28 31 32) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regc1 "reg2port")))
(make-edge :value 'e29
           :type 'data
           :in-node (nth 31 *node-list*)
           :out-node nil
           :attribute (pairlis '(storage allocation implement)
                               '(reg regc2 "reg2port")))
;; 30
(make-edge :value 'c[n-2]
           :type 'data
           :in-node nil
           :out-node (my-nth '(23 24 26 33) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regc2 "reg2port")))
(make-edge :value 'e31
           :type 'data
           :in-node (nth 32 *node-list*)
           :out-node (my-nth '(33) *node-list*)
           :attribute (pairlis '(storage)
                               '(var)))
(make-edge :value 'e32
           :type 'data
           :in-node (nth 33 *node-list*)
           :out-node (my-nth '(34) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regc3 "reg2port")))
```

```
(make-edge :value 'e33
           :type 'data
           :in-node (nth 27 *node-list*)
           :out-node (my-nth '(28) *node-list*)
           :attribute (pairlis '(storage)
                               '(var)))
(make-edge :value 'e34
           :type 'data
           :in-node (nth 28 *node-list*)
           :out-node (my-nth '(29) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regc4 "reg2port")))
(make-edge :value 'e35
           :type 'data
           :in-node (nth 23 *node-list*)
           :out-node (my-nth '(24) *node-list*)
           :attribute (pairlis '(storage)
                               '(var)))
(make-edge :value 'e36
           :type 'data
           :in-node (nth 24 *node-list*)
           :out-node (my-nth '(25) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regc4 "reg2port")))
(make-edge :value 'e37
           :type 'data
           :in-node (nth 25 *node-list*)
           :out-node (my-nth '(26) *node-list*)
           :attribute (pairlis '(storage)
                               '(var)))
(make-edge :value 'e38
           :type 'data
           :in-node (nth 26 *node-list*)
           :out-node (my-nth '(29) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regc3 "reg2port")))
(make-edge :value 'e39
           :type 'data
           :in-node (nth 29 *node-list*)
           :out-node (my-nth '(30) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regc4 "reg2port")))

;; 40
(make-edge :value 'c
           :type 'data
           :in-node (nth 30 *node-list*)
           :out-node (my-nth '(34) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regc1 "reg2port")))
```

```
(make-edge :value 'IN4
           :type 'data
           :in-node (nth 34 *node-list*)
           :out-node (my-nth '(39) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regI4 "reg2port")))
(make-edge :value 'd[n-1]
           :type 'data
           :in-node nil
           :out-node (my-nth '(35 36 38 40) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regd1 "reg2port")))
(make-edge :value 'e43
           :type 'data
           :in-node (nth 35 *node-list*)
           :out-node (my-nth '(36) *node-list*)
           :attribute (pairlis '(storage)
                               '(var)))
(make-edge :value 'e44
           :type 'data
           :in-node (nth 36 *node-list*)
           :out-node (my-nth '(37) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regd3 "reg2port")))
(make-edge :value 'e45
           :type 'data
           :in-node (nth 37 *node-list*)
           :out-node (my-nth '(38) *node-list*)
           :attribute (pairlis '(storage)
                               '(var)))
(make-edge :value 'e46
           :type 'data
           :in-node (nth 38 *node-list*)
           :out-node (my-nth '(39) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regd3 "reg2port")))
(make-edge :value 'd
           :type 'data
           :in-node (nth 39 *node-list*)
           :out-node (my-nth '(41) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regd1 "reg2port")))
(make-edge :value 'e48
           :type 'data
           :in-node (nth 40 *node-list*)
           :out-node (my-nth '(41) *node-list*)
           :attribute (pairlis '(storage allocation implement)
                               '(reg regd2 "reg2port")))
```

```
       (make-edge :value 'out
                  :type 'data
                  :in-node (nth 41 *node-list*)
                  :out-node '(parent)
                  :attribute (pairlis '(storage allocation implement)
                                      '(reg regout "reg2port")))
       ))

;;;;;;;;;;;;;;;; edge width and parent node ;;;;;;;;;;;;;;;;;;;;

(dolist (i *edge-list*)
        (setf (edge-width i) 32)
        (setf (edge-attribute i)
              (acons 'parent (nth 0 *node-list*)
                     (edge-attribute i))))

;;;;;;;;;;;;;;;; subgraph ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setf (node-subgraph (nth 0 *node-list*))
      (list (cdr *node-list*)
            *edge-list*
            (my-nth '(0) *edge-list*)
            (my-nth '(49) *edge-list*)))

;;;;;;;;;;;;;;;;;; in-edge of nodes ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setf (node-in-edge (nth 1 *node-list*))
  (my-nth '(1 10) *edge-list*))
(setf (node-in-edge (nth 2 *node-list*))
  (my-nth '(3) *edge-list*))
(setf (node-in-edge (nth 3 *node-list*))
  (my-nth '(3 6) *edge-list*))
(setf (node-in-edge (nth 4 *node-list*))
  (my-nth '(7) *edge-list*))
(setf (node-in-edge (nth 5 *node-list*))
  (my-nth '(3 8) *edge-list*))
(setf (node-in-edge (nth 6 *node-list*))
  (my-nth '(9 13) *edge-list*))
(setf (node-in-edge (nth 7 *node-list*))
  (my-nth '(4) *edge-list*))
(setf (node-in-edge (nth 8 *node-list*))
  (my-nth '(4 11) *edge-list*))
(setf (node-in-edge (nth 9 *node-list*))
  (my-nth '(12) *edge-list*))
(setf (node-in-edge (nth 10 *node-list*))
  (my-nth '(3) *edge-list*))
(setf (node-in-edge (nth 11 *node-list*))
  (my-nth '(3 4) *edge-list*))
(setf (node-in-edge (nth 12 *node-list*))
  (my-nth '(2 14) *edge-list*))
(setf (node-in-edge (nth 13 *node-list*))
  (my-nth '(16) *edge-list*))
```

```
(setf (node-in-edge (nth 14 *node-list*))
  (my-nth '(19 16) *edge-list*))
(setf (node-in-edge (nth 15 *node-list*))
  (my-nth '(20 22) *edge-list*))
(setf (node-in-edge (nth 16 *node-list*))
  (my-nth '(17) *edge-list*))
(setf (node-in-edge (nth 17 *node-list*))
  (my-nth '(21 17) *edge-list*))
(setf (node-in-edge (nth 18 *node-list*))
  (my-nth '(15 23) *edge-list*))
(setf (node-in-edge (nth 19 *node-list*))
  (my-nth '(16) *edge-list*))
(setf (node-in-edge (nth 20 *node-list*))
  (my-nth '(16) *edge-list*))
(setf (node-in-edge (nth 21 *node-list*))
  (my-nth '(17 25) *edge-list*))
(setf (node-in-edge (nth 22 *node-list*))
  (my-nth '(24 26) *edge-list*))
(setf (node-in-edge (nth 23 *node-list*))
  (my-nth '(30) *edge-list*))
(setf (node-in-edge (nth 24 *node-list*))
  (my-nth '(35 30) *edge-list*))
(setf (node-in-edge (nth 25 *node-list*))
  (my-nth '(36) *edge-list*))
(setf (node-in-edge (nth 26 *node-list*))
  (my-nth '(37 30) *edge-list*))
(setf (node-in-edge (nth 27 *node-list*))
  (my-nth '(28) *edge-list*))
(setf (node-in-edge (nth 28 *node-list*))
  (my-nth '(33 28) *edge-list*))
(setf (node-in-edge (nth 29 *node-list*))
  (my-nth '(34 38) *edge-list*))
(setf (node-in-edge (nth 30 *node-list*))
  (my-nth '(39 27) *edge-list*))
(setf (node-in-edge (nth 31 *node-list*))
  (my-nth '(28) *edge-list*))
(setf (node-in-edge (nth 32 *node-list*))
  (my-nth '(28) *edge-list*))
(setf (node-in-edge (nth 33 *node-list*))
  (my-nth '(30 31) *edge-list*))
(setf (node-in-edge (nth 34 *node-list*))
  (my-nth '(40 32) *edge-list*))
(setf (node-in-edge (nth 35 *node-list*))
  (my-nth '(42) *edge-list*))
(setf (node-in-edge (nth 36 *node-list*))
  (my-nth '(43 42) *edge-list*))
(setf (node-in-edge (nth 37 *node-list*))
  (my-nth '(44) *edge-list*))
(setf (node-in-edge (nth 38 *node-list*))
  (my-nth '(42 45) *edge-list*))
```

```
(setf (node-in-edge (nth 39 *node-list*))
  (my-nth '(41 46) *edge-list*))
(setf (node-in-edge (nth 40 *node-list*))
  (my-nth '(42) *edge-list*))
(setf (node-in-edge (nth 41 *node-list*))
  (my-nth '(47 48) *edge-list*))
(setf (node-in-edge (nth 42 *node-list*))
  (my-nth '(0) *edge-list*))

;;;;;;;;;;;;;;;;;;; out-edge of nodes ;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setf (node-out-edge (nth 1 *node-list*))
  (my-nth '(2) *edge-list*))
(setf (node-out-edge (nth 2 *node-list*))
  (my-nth '(6) *edge-list*))
(setf (node-out-edge (nth 3 *node-list*))
  (my-nth '(7) *edge-list*))
(setf (node-out-edge (nth 4 *node-list*))
  (my-nth '(8) *edge-list*))
(setf (node-out-edge (nth 5 *node-list*))
  (my-nth '(9) *edge-list*))
(setf (node-out-edge (nth 6 *node-list*))
  (my-nth '(10) *edge-list*))
(setf (node-out-edge (nth 7 *node-list*))
  (my-nth '(11) *edge-list*))
(setf (node-out-edge (nth 8 *node-list*))
  (my-nth '(12) *edge-list*))
(setf (node-out-edge (nth 9 *node-list*))
  (my-nth '(13) *edge-list*))
(setf (node-out-edge (nth 10 *node-list*))
  (my-nth '(5) *edge-list*))
(setf (node-out-edge (nth 11 *node-list*))
  (my-nth '(14) *edge-list*))
(setf (node-out-edge (nth 12 *node-list*))
  (my-nth '(15) *edge-list*))
(setf (node-out-edge (nth 13 *node-list*))
  (my-nth '(19) *edge-list*))
(setf (node-out-edge (nth 14 *node-list*))
  (my-nth '(20) *edge-list*))
(setf (node-out-edge (nth 15 *node-list*))
  (my-nth '(23) *edge-list*))
(setf (node-out-edge (nth 16 *node-list*))
  (my-nth '(21) *edge-list*))
(setf (node-out-edge (nth 17 *node-list*))
  (my-nth '(22) *edge-list*))
(setf (node-out-edge (nth 18 *node-list*))
  (my-nth '(24) *edge-list*))
(setf (node-out-edge (nth 19 *node-list*))
  (my-nth '(18) *edge-list*))
(setf (node-out-edge (nth 20 *node-list*))
  (my-nth '(25) *edge-list*))
```

```
(setf (node-out-edge (nth 21 *node-list*))
  (my-nth '(26) *edge-list*))
(setf (node-out-edge (nth 22 *node-list*))
  (my-nth '(27) *edge-list*))
(setf (node-out-edge (nth 23 *node-list*))
  (my-nth '(35) *edge-list*))
(setf (node-out-edge (nth 24 *node-list*))
  (my-nth '(36) *edge-list*))
(setf (node-out-edge (nth 25 *node-list*))
  (my-nth '(37) *edge-list*))
(setf (node-out-edge (nth 26 *node-list*))
  (my-nth '(38) *edge-list*))
(setf (node-out-edge (nth 27 *node-list*))
  (my-nth '(33) *edge-list*))
(setf (node-out-edge (nth 28 *node-list*))
  (my-nth '(34) *edge-list*))
(setf (node-out-edge (nth 29 *node-list*))
  (my-nth '(39) *edge-list*))
(setf (node-out-edge (nth 30 *node-list*))
  (my-nth '(40) *edge-list*))
(setf (node-out-edge (nth 31 *node-list*))
  (my-nth '(29) *edge-list*))
(setf (node-out-edge (nth 32 *node-list*))
  (my-nth '(31) *edge-list*))
(setf (node-out-edge (nth 33 *node-list*))
  (my-nth '(32) *edge-list*))
(setf (node-out-edge (nth 34 *node-list*))
  (my-nth '(41) *edge-list*))
(setf (node-out-edge (nth 35 *node-list*))
  (my-nth '(43) *edge-list*))
(setf (node-out-edge (nth 36 *node-list*))
  (my-nth '(44) *edge-list*))
(setf (node-out-edge (nth 37 *node-list*))
  (my-nth '(45) *edge-list*))
(setf (node-out-edge (nth 38 *node-list*))
  (my-nth '(46) *edge-list*))
(setf (node-out-edge (nth 39 *node-list*))
  (my-nth '(47) *edge-list*))
(setf (node-out-edge (nth 40 *node-list*))
  (my-nth '(48) *edge-list*))
(setf (node-out-edge (nth 41 *node-list*))
  (my-nth '(49) *edge-list*))
(setf (node-out-edge (nth 42 *node-list*))
  (my-nth '(1) *edge-list*))

;;;;;;;;;;;;;;;;;;;;;; ordering ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setf (node-attribute (nth 1 *node-list*))
  (acons 'order 5 (node-attribute (nth 1 *node-list*))))
(setf (node-attribute (nth 2 *node-list*))
  (acons 'order 0 (node-attribute (nth 2 *node-list*))))
```

```
(setf (node-attribute (nth 3 *node-list*))
  (acons 'order 0 (node-attribute (nth 3 *node-list*))))
(setf (node-attribute (nth 4 *node-list*))
  (acons 'order 1 (node-attribute (nth 4 *node-list*))))
(setf (node-attribute (nth 5 *node-list*))
  (acons 'order 1 (node-attribute (nth 5 *node-list*))))
(setf (node-attribute (nth 6 *node-list*))
  (acons 'order 3 (node-attribute (nth 6 *node-list*))))
(setf (node-attribute (nth 7 *node-list*))
  (acons 'order 2 (node-attribute (nth 7 *node-list*))))
(setf (node-attribute (nth 8 *node-list*))
  (acons 'order 2 (node-attribute (nth 8 *node-list*))))
(setf (node-attribute (nth 9 *node-list*))
  (acons 'order 3 (node-attribute (nth 9 *node-list*))))
(setf (node-attribute (nth 10 *node-list*))
  (acons 'order 4 (node-attribute (nth 10 *node-list*))))
(setf (node-attribute (nth 11 *node-list*))
  (acons 'order 4 (node-attribute (nth 11 *node-list*))))
(setf (node-attribute (nth 12 *node-list*))
  (acons 'order 6 (node-attribute (nth 12 *node-list*))))
(setf (node-attribute (nth 13 *node-list*))
  (acons 'order 0 (node-attribute (nth 13 *node-list*))))
(setf (node-attribute (nth 14 *node-list*))
  (acons 'order 0 (node-attribute (nth 14 *node-list*))))
(setf (node-attribute (nth 15 *node-list*))
  (acons 'order 2 (node-attribute (nth 15 *node-list*))))
(setf (node-attribute (nth 16 *node-list*))
  (acons 'order 1 (node-attribute (nth 16 *node-list*))))
(setf (node-attribute (nth 17 *node-list*))
  (acons 'order 1 (node-attribute (nth 17 *node-list*))))
(setf (node-attribute (nth 18 *node-list*))
  (acons 'order 5 (node-attribute (nth 18 *node-list*))))
(setf (node-attribute (nth 19 *node-list*))
  (acons 'order 4 (node-attribute (nth 19 *node-list*))))
(setf (node-attribute (nth 20 *node-list*))
  (acons 'order 3 (node-attribute (nth 20 *node-list*))))
(setf (node-attribute (nth 21 *node-list*))
  (acons 'order 3 (node-attribute (nth 21 *node-list*))))
(setf (node-attribute (nth 22 *node-list*))
  (acons 'order 6 (node-attribute (nth 22 *node-list*))))
(setf (node-attribute (nth 23 *node-list*))
  (acons 'order 0 (node-attribute (nth 23 *node-list*))))
(setf (node-attribute (nth 24 *node-list*))
  (acons 'order 0 (node-attribute (nth 24 *node-list*))))
(setf (node-attribute (nth 25 *node-list*))
  (acons 'order 1 (node-attribute (nth 25 *node-list*))))
(setf (node-attribute (nth 26 *node-list*))
  (acons 'order 1 (node-attribute (nth 26 *node-list*))))
(setf (node-attribute (nth 27 *node-list*))
  (acons 'order 2 (node-attribute (nth 27 *node-list*))))
```

```
(setf (node-attribute (nth 28 *node-list*))
  (acons 'order 2 (node-attribute (nth 28 *node-list*))))
(setf (node-attribute (nth 29 *node-list*))
  (acons 'order 3 (node-attribute (nth 29 *node-list*))))
(setf (node-attribute (nth 30 *node-list*))
  (acons 'order 5 (node-attribute (nth 30 *node-list*))))
(setf (node-attribute (nth 31 *node-list*))
  (acons 'order 4 (node-attribute (nth 31 *node-list*))))
(setf (node-attribute (nth 32 *node-list*))
  (acons 'order 4 (node-attribute (nth 32 *node-list*))))
(setf (node-attribute (nth 33 *node-list*))
  (acons 'order 4 (node-attribute (nth 33 *node-list*))))
(setf (node-attribute (nth 34 *node-list*))
  (acons 'order 6 (node-attribute (nth 34 *node-list*))))
(setf (node-attribute (nth 35 *node-list*))
  (acons 'order 0 (node-attribute (nth 35 *node-list*))))
(setf (node-attribute (nth 36 *node-list*))
  (acons 'order 0 (node-attribute (nth 36 *node-list*))))
(setf (node-attribute (nth 37 *node-list*))
  (acons 'order 1 (node-attribute (nth 37 *node-list*))))
(setf (node-attribute (nth 38 *node-list*))
  (acons 'order 1 (node-attribute (nth 38 *node-list*))))
(setf (node-attribute (nth 39 *node-list*))
  (acons 'order 3 (node-attribute (nth 39 *node-list*))))
(setf (node-attribute (nth 40 *node-list*))
  (acons 'order 2 (node-attribute (nth 40 *node-list*))))
(setf (node-attribute (nth 41 *node-list*))
  (acons 'order 4 (node-attribute (nth 41 *node-list*))))
(setf (node-attribute (nth 42 *node-list*))
  (acons 'order 4 (node-attribute (nth 42 *node-list*))))
```

C.11 Conclusion

A hierarchical control/data flow graph format in the Common Lisp syntax is described. Although this format is mainly designed for the input of the HYPER hardware mapper, it is general enough for many other signal flow graph descriptions. This format contains six sections of descriptions – node-list, edge-list parent-node, subgraph, input/output edge of nodes, and control steps. Each section has been described in detail in this appendix. Flow graphs of the proposed format have been used to describe a 7th order IIR filter, the Viterbi processor, the Epsilon processor and many other real examples for synthesis. All the examples have been run through the hardware mapping process and the required sdl files and bds files for Lager IV have been generated. More features of the format may be included in the future as we experiment more application examples. As the format is standing right now, it is general enough to handle most of the applications that DSP designers are interested in.