A HARDWARE MAPPER FOR THE HYPER HIGH LEVEL SYNTHESIS SYSTEM

by

Ole Bentz

Memorandum No. UCB/ERL M96/97

15 December 1996

A HARDWARE MAPPER FOR THE HYPER HIGH LEVEL SYNTHESIS SYSTEM

by

Ole Bentz

Memorandum No. UCB/ERL M96/97

15 December 1996

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A HARDWARE MAPPER FOR THE HYPER HIGH LEVEL SYNTHESIS SYSTEM

by

Ole Bentz

ABSTRACT

The goal of this project is to develop an interface between a high level synthesis tool and several layout generators. The interface, called a "Hardware Mapper", is a complete program which can translate a synthesized design to two different layout generator languages.

The major challenge of the research is to derive a complete application specific integrated circuit (ASIC) description from a synthesized design. The description should be generic and independent of the targeted layout generator language, such that new languages can easily be accommodated. Another challenge is to translate the generic description to specific languages in a manner that enables layout generators to achieve area efficient layouts.

The layouts that result from using the Hardware Mapper on a set of benchmarks demonstrate its capability to generate design specifications such that efficient layouts can be generated automatically.

ACKNOWLEDGEMENTS

First of all, I would like to thank my research advisor Jan Rabaey for his generous support and encouragement of my work. Working on a part of the Hyper high level synthesis project has been very rewarding. Thank you, Jan, for your advice, your patience and your guidance.

I would also like to thank Shan-Hsi Sean Huang for all his help, especially with the control generation, and the many hours of fruitful discussions about various Hyper related issues.

Also, Brian Richards has been a constant source of help with his in-depth knowledge of the "features" of the Lager tools. Thank you, Brian.

To all my other colleagues in the "BJGroup": it has been great to work with you. Thanks for the various ways each one of you has contributed to my academic growth.

Lastly, I would like to dedicate this thesis to my wife, Kelly, who I love more than anything in this world. Kelly, your patience and love for me knows no bounds. Thank you for hanging in there with me.

TABLE OF CONTENTS

1 INTRODUCTION

The task of implementing application specific integrated circuits (ASICs) from behavioral descriptions involves many different areas of expertise. These areas range from algorithm development and manipulation to chip layout and verification. In recent years, much effort has been put into capturing the expertise of ASIC designers such as to automate the bulk of their work. There appears to be a division of the specialty areas into two main categories. The first category deals with the more abstract tasks of taking an algorithm, transforming it if needed, assigning operations to suitable units and developing a schedule for the operations that share units. This category is known as "high level synthesis". The second category covers the rest of the tasks, namely the placement of units and the routing of wiring between units. The tools in this category are known as "layout generators" (or "silicon compilers").

High level synthesis tools and layout generation tools deal with very different issues of the ASIC design. Consequently, data formats to represent designs are specific to individual tools and it can be hard to transfer designs from high level tools to layout tools. Even beyond different data formats, there is often a gap between high level synthesis tools and layout tools, because the former is considered "finished" when a design has been "scheduled", but the latter expects a completely specified design.

Hardware Mapping is the process that interfaces high level synthesis tools with layout generation tools. A *Hardware Mapper* is a tool that derives a basic implementation of a design from a high level synthesis data representation, adorns the implementation with necessary details, and presents a fully specified design to a layout generator. The format in which a design is presented to a layout generator is normally called a "hardware description language".

A *Hardware Mapper* has been implemented for the Hyper high level synthesis system. The *Mapper* fills the gap between Hyper and layout generators (see Figure 1.1). It is capable of mapping Hyper designs to two different hardware description languages. The two languages supported are VHDL, the IEEE standardized hardware description language, and SDL, the structural description language used by the Lager layout generation tools.

The rest of this report will describe the details of the *Hardware Mapper*. Chapter 2 gives a brief overview of Hyper, as well as an overview of the *Hardware Mapper*. Chapter 3 outlines the resources which the *Hardware Mapper* uses at run-time. Chapter 4 explains how the *Mapper* extracts design information from the Hyper database, and what further design detail it generates. Chapter 5 presents the mapping to the VHDL language, and chapter 6 shows the mapping to the SDL language. Chapter 7 gives some examples of *Hardware Mapper* generated layouts. Chapter 8 offers some conclusions. and suggestions for future work.

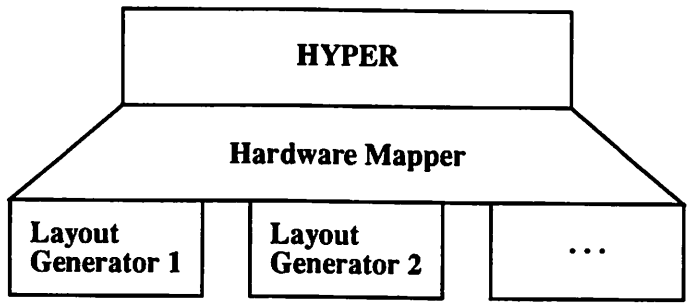

FIGURE 1.1 : The Hardware Mapper as an interface

2 OVERVIEW

2.1 The Hyper System

Hyper is an interactive menu-driven design environment for synthesizing high-performance digital ASICs. It consists of a number of software modules linked under a common X-windows based management tool called Xhyper. Figure 2.1 shows the Xhyper window. All Hyper's software modules communicate through a common database as shown in Figure 2.2.

The input to Hyper is a Silage description of an algorithm. Silage is an applicative signal-flowgraph language that is well suited for describing digital systems that have little explicit control.

A Silage description is parsed and translated into a control/data flowgraph.The flowgraph represents the same information as the original Silage description, but it is easier to manipulate. Also, a flowgraph can be adorned with various details which can not be represented in the Silage description.

The selection step assigns a unit (or a group of units) from a hardware library to each type of node in the flowgraph. The hardware library contains a number of different implementations of certain blocks in order to trade off performance, area and power.

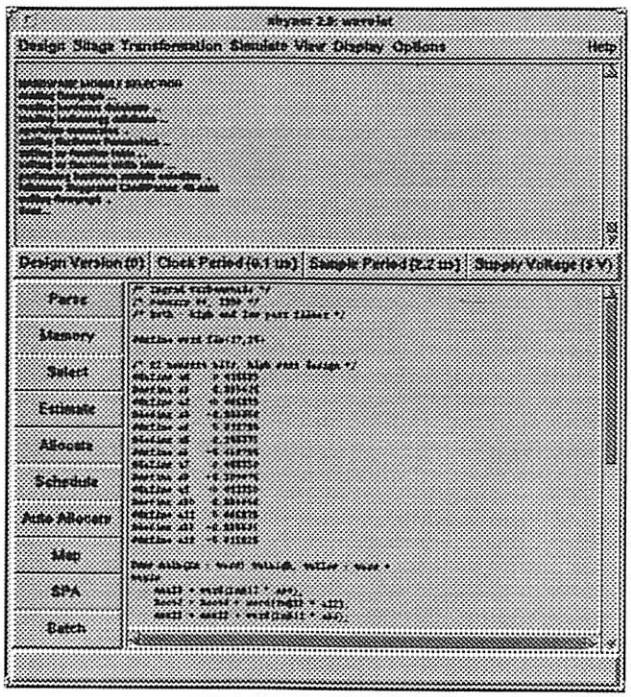

FIGURE 2.1 : The Xhyper window.

The transformation phase consists of algorithmic transformations which can be performed on the flowgraph in order to achieve some specific result. Some possible transformations include expansion of fixed-coefficient multiplications into a series of adds and shifts, algebraic transformations, loop unrolling, retiming and pipelining.

The estimation stage provides the designer with estimated resource requirements. The resources estimated are: the number of cycles in the critical path, the number and type of execution units (EXUs), the number of registers associated with each EXU, and the number of buses connecting execution units to register files.

The assignment/scheduling step assigns each flowgraph node to a specific hardware unit at a certain time. The number of each type of hardware units that are needed is derived. Prior to scheduling the clock period and sampling rates must be specified in order to define the number of cycles available to the system.

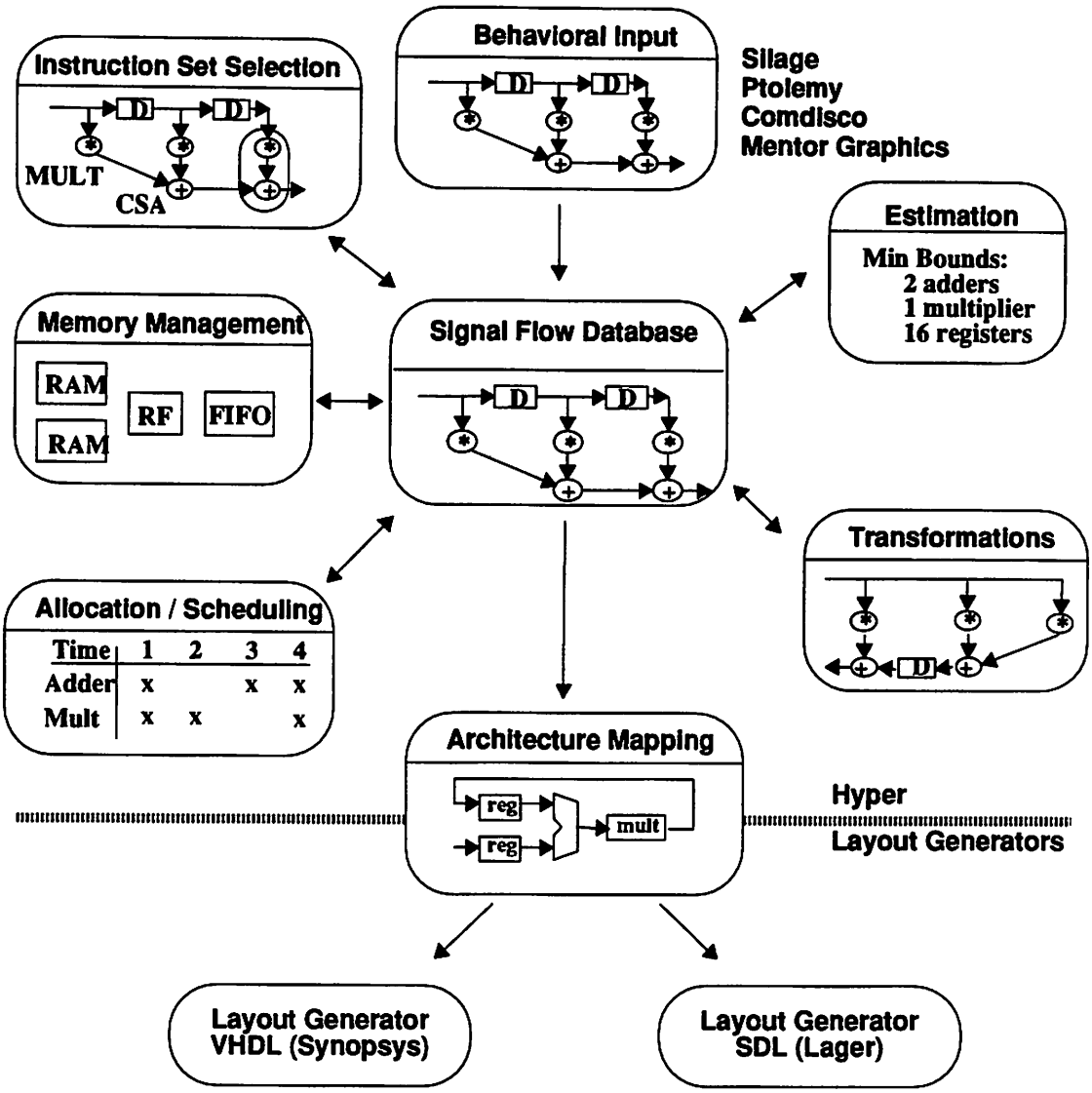

FIGURE 2.2 : The Hyper modules.

The architecture mapping, also called hardware mapping, maps the flowgraph onto the selected hardware units by generating the interconnect information, placing multiplexers and buffers where needed, defining the finite state machine and control logic, and partitioning the datapath. After this step the system is completely defined.

The output of the hardware mapper is a group of hardware description files. They can be requested in either VHDL format [VHD87], IEEE's standardized hardware description language, or in the SDL format [Bro92], the Structural Description Language used for the LagerIV silicon compiler [Lager91].

The focus of this report is the hardware mapping step. The procedures and methodologies that are used will be discussed in detail.

2.2 Hardware Mapper Overview

The *Hardware Mapper* has two main phases: a target independent phase and a target specific phase (Figure 2.3). During the target independent phase, the *Mapper* extracts design information from a Hyper generated flowgraph and adds various details. For example, buffers and multiplexers are added, and the control specification is derived. This phase is generic and it is used regardless of the targeted hardware description language. Chapter 4 explains this process in detail. There are two target specific phases, each of which generates a different format, either VHDL or SDL. Although these two phases have many similarities, they are different enough to need

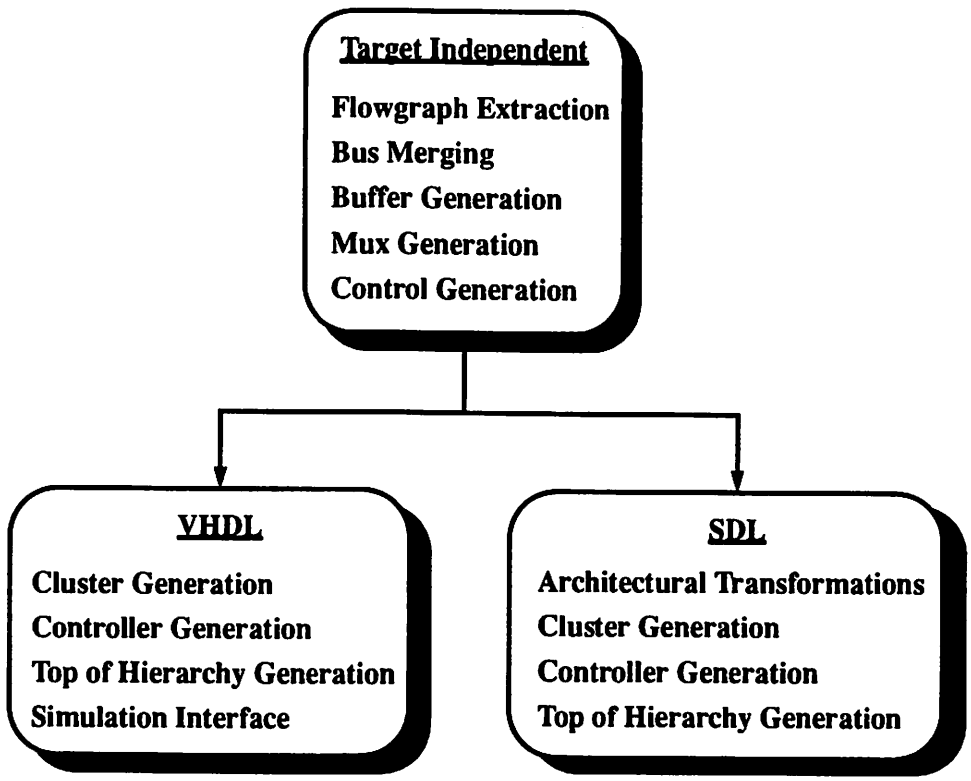

FIGURE 2.3 : The Hardware Mapper's main phases.

separate discussion. Chapter 5 presents the VHDL phase, and Chapter 6 describes the SDL phase.

3 RESOURCES

High level synthesis involves a phase in which it is decided how all the operations in an algorithm will be performed. This normally involves a matching process in which an operation (e.g. an addition) is matched with a pre-designed hardware unit which is capable of performing that operation. In order to expedite the matching process, it is common to organize hardware units into libraries. An index that lists all the units in a library, including which operations they are capable of performing, can then be used to match operations with hardware units. Within one library there may be many units that can perform the same operation. In that case, it is necessary to select which one of the possible choices will be best to use. For example, if a designer considers speed most important, the fastest unit can be chosen. Otherwise, the smallest unit or the most power efficient unit can be selected.

There are many aspects of a hardware library that are not important for a high level synthesis tool to have access to. Therefore, it is beneficial to extract the necessary information and keep it in a library "data sheet". A library "data sheet" has to be organized such that units that can perform the same operation are listed together. It should also have information about size, speed or power consumption to help choosing the best units. In addition, there needs to be information about how a unit can be connected with other units. If a unit can do several operations, it is necessary to tell it

when to do one operation and when to do another. That type of information should also be listed in the data sheet.

Hyper uses the library "data sheet" approach to organize relevant information about hardware units. In fact, the data sheet is the only view Hyper has of a library, and it is often just referred to as "the library". A library (using the Hyper terminology) typically lists enough different units to allow all common operations to be matched with at least one of them. In addition to a data sheet, Hyper has a facility for describing information that relates to all the hardware units that are listed in the library. For example, if all the hardware units in the library are supposed to be connected to a power supply of 3 volts (V), then it can be specified just once. In Hyper terminology, the facility that contains this type of information is called a "technology" file.

The following two sections outline the interactions the *Hardware Mapper* has with the library and the technology file.

3.1 Hardware Library

The *Hardware Mapper* uses a library "data sheet" to get information about hardware units. For example, the size of a unit is needed when the *Mapper* predicts the size of a chip design. The *Mapper* also needs to find out how a unit should be connected with other units. For that task, the names of inputs and outputs have to be determined from the library.

The *Hardware Mapper* can access a library through a set of programs. These programs are also used by other tools in Hyper, so they are not formally a part of the *Mapper*'s routines. However, they are used frequently enough to warrant some discussion. Most of the access programs are "question and answer" types, which means they provide information from the library upon request.

Table 3.1 shows the routines that are used to access the library. Each of the entries in the table have a "question" column that, in plain english, requests information. The table also lists the actual program names and their arguments.

Question	Routine	Arguments	Description
Where is the cell?	HwGetCell	Function, Name	Get a pointer to the unit which is called *Name* and is listed in the group *Function.*
What are the input or output names of a cell?	HwGetData	Function, Name, Parameters, Direction	Get the names of the inputs or outputs of a unit which is called *Name* and is listed in the group *Function.*
What are the names of all the control signals of a cell?	HwGetControl	Cell, Parameters, Direction	Get all the names of the input or output control signals of a unit, including their values (if any).
On what side of a unit are the control terminals located?	HwGetControlEdge	Cell, Parameters	Get all the control signals' location.
What parameters are required to configure a unit?	HwGetHw-Parameters	Cell, Parameters	Get all the hardware parameters that are required by a cell.
What is the name of the SDL file for a unit?	HwGetSdlName	Function, Name	Get the name of the VHDL file for a unit which is known to Hyper as *Name* and is listed in the group *Function.*
What is the name of the VHDL file for a unit?	HwGetVhdlName	Function, Name	Get the name of the VHDL file for a unit which is known to Hyper as *Name* and is listed in the group *Function.*
What is the smallest unit that can perform the operation *Function?*	HwGetCheapestCell	Function, Parameters	Get a pointer to the smallest unit in the group *Function.*

Table 3.1 : Library Access Routines

Question	Routine	Arguments	Description
What is the size of a unit?	HwGetArea	Function, Name, Parameters	Get the area of a unit called *Name* which is listed in the group *Function.*
What is the value of...?	HwGetGeneric	Function, Name, Parameters, Keyword	Get the value of *Keyword* for a unit called *Name* which is listed in the group *Function.*
Is there are unit in the library that can do *Function?*	HwIsFunctionIn-Library	Function	
Which layout generation methodology is used for this unit?	HwGetFunction-Source	Function	There are three possible answers: datapath, array, standard cell. See below.

Table 3.1 : Library Access Routines

Hyper's library indices are split into three separate portions. This is a feature that was introduced to help discern between units that needed different layout generation approaches in the Lager layout generation system. Lager has three such approaches, including: datapath, array and standard cell. It is not important at this stage to understand these different approaches, but it does explain why the function "HwGetFunctionSource" is used.

3.2 Technology File

A "technology file" contains information which applies to all the hardware units listed in a library. Thus, in order to properly understand the information which can be obtained from the library, it must be combined with the technology file's data. The parts of the technology file that the *Mapper* needs to access are listed in table 3.2.

Item	Sample Value	Description
library	low_power	This identifies the library.
nr_of_clocks	2	The number of clock phases in a design.
size_units	micron	All geometries are given in microns.
size_scale	(/ 6 10)	All geometries should be scaled by 6/10 = 0.6. This is specified because the syntax does not allow floating point numbers.
state_transition	falling	The finite state machine in the controller changes state on this edge of the clock.

Table 3.2 : Technology File

4

TARGET INDEPENDENT PHASE

The first phase of the *Hardware Mapper* is the target independent phase. During this phase the *Mapper* translates a flowgraph into an intermediate representation which contains adequate information to allow mapping onto many different target platforms.

The flow of the target independent phase is shown in Figure 4.1. There are five steps in this phase, but the middle step (bus merging) may not always be executed. Before presenting all the details of the five steps, section 4.1 will discuss the underlying architecture model. Once that foundation has been laid, section 4.2 will explain the register selection process. Section 4.3 will elaborate on the basic translation process from a flowgraph to the intermediate representation. Section 4.4 gives details of the bus merging routines. Section 4.5 explains how buffers and multiplexers are added to the design. Finally, Section 4.6 describes the way control is generated.

4.1 ARCHITECTURE MODEL

Hyper makes certain assumptions about the architectural style that is used for design implementations. Design decisions are simplified for all the tools in Hyper, since some aspects of the architecture are fixed. Before going into details about the mapping

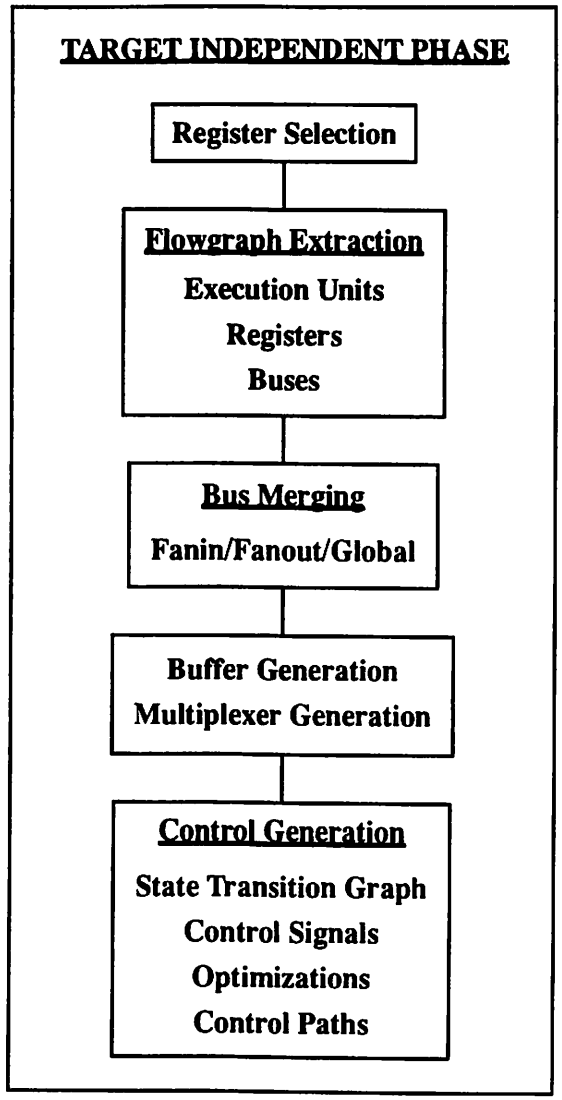

FIGURE 4.1 : The target independent phase

process it is useful to describe the model, since many components of the *Mapper* are artifacts of the architecture.

The Hyper architecture model contains execution units (EXUs) and register files, as well as a crossbar interconnect network. The model does not limit the number of units in the architecture, but physical limitations (such as maximum chip size) will naturally dictate an upper bound. Figure 4.2 shows the basic architecture. The size of register files is not limited by the model, but, again, there are physical limits. The crossbar interconnect network provides connections from any execution unit to any

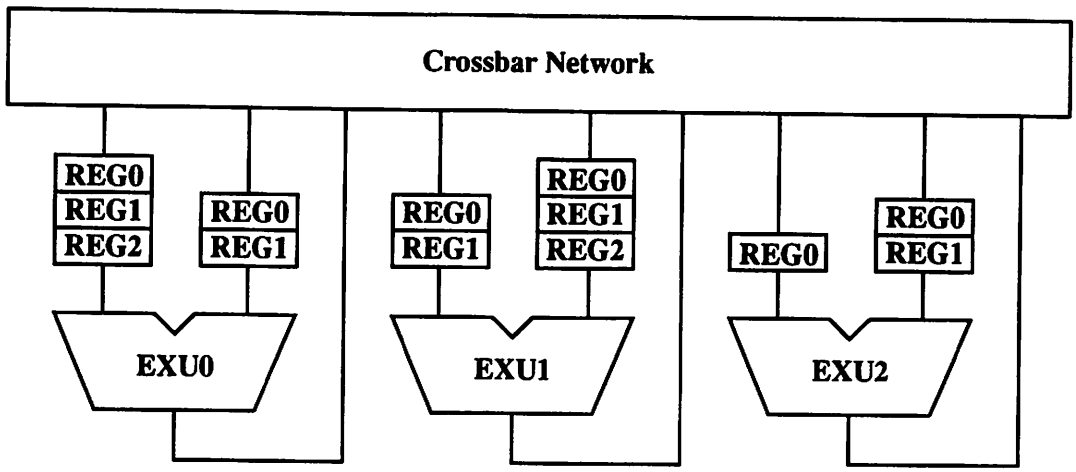

FIGURE 4.2 : Hyper's basic architecture model

register file. This network may form different connections at different times. All register files have a single input port, such that only one value can be written at a time. There is one dedicated bus between the output of a register file and the input of the execution unit it "belongs" to.

The crossbar network is implemented by a set of dedicated buses, bus drivers and multiplexers. A bus driver is a buffer that can drive the output of an execution unit unto a bus. A multiplexer is capable of selecting one bus from a given set of buses. The crossbar network is configured by choosing which bus drivers are active, and which multiplexer inputs are selected. Figure 4.3 shows the implementation of a crossbar network.

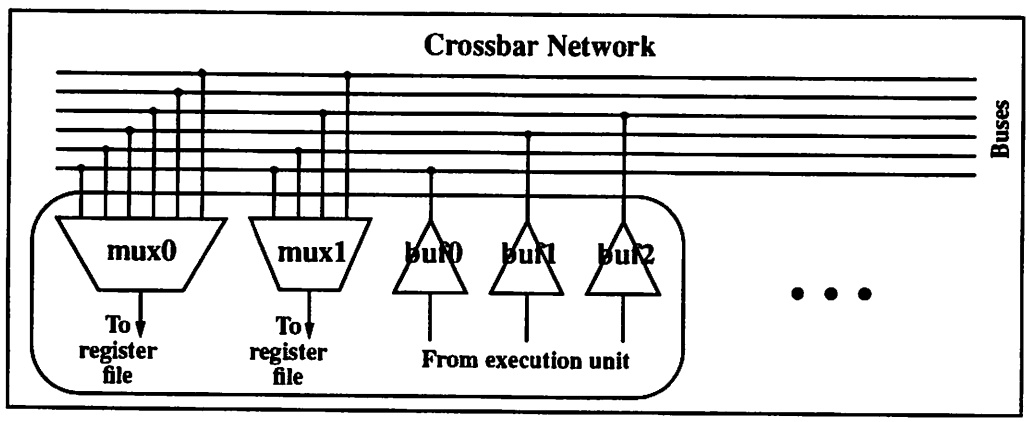

FIGURE 4.3 : Hyper's implementation of a crossbar network.

The refined Hyper architecture model emerges as the basic architecture is merged with the crossbar implementation. The multiplexers and bus drivers are grouped with the register files or execution units that they are associated with. This is shown in Figure 4.4. There are dedicated buses between multiplexers and register files, and between execution units and bus drivers.

In the rest of this report, the term "*cluster*" refers to a group of multiplexers, register files, execution units and buffers that are connected only by dedicated buses. Clusters can communicate with each other through the global bus network. A cluster typically has one execution unit, one or two register files and a few muxes and buffers. However, there is no limit to the number of units that are allowed in a cluster.

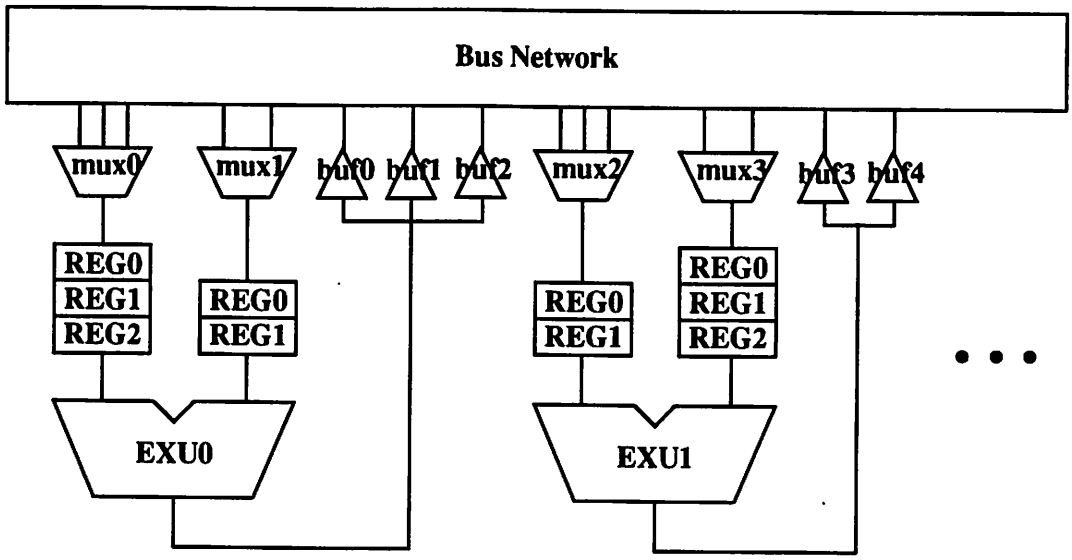

FIGURE 4.4 : Hyper's refined architecture model

4.2 REGISTER SELECTION

The first step of the target independent phase is the register selection step. During this step, the *Hardware Mapper* selects which registers from the library will be used for each of the register files in the design. This selection task would naturally be

the duty of the module selection routines in Hyper. However, at the time when hardware module selection is performed, there is not enough information available to make the optimal choice. Therefore, the *Hardware Mapper* is entrusted with the task of selecting the most suitable types of registers. The *Mapper* considers only area when selecting the register hardware.

4. 2. 1 Register Concepts

Hyper has a single concept of foreground storage: register files. In the Hyper terminology, a register file has a single input and a single output. The bit width can be specified and it can contain constant values (read-only) or variables (read-write). To provide proper implementations of this concept, the *Mapper* has a more differentiated vision which includes four types of registers. They include register files, single registers with tri-stated outputs, single registers without tri-stated outputs and flip-flops. The first three categories are thought of as being parameterizable in the bit width, while the flip-flop is thought of as a single bit standard cell block. The following paragraphs outline the assumptions and restrictions inherent to these register categories.

Register Files

Register files contain one or more registers, which could be either read/write or read only. It can address one location for reading and one for writing in each clock cycle. All the registers in the file have the same bit width, which is at least one bit.

Registers with tri-statable outputs

These registers can store one value at a time, and it can be either read/write or read only. It can both be read and written in a clock cycle. The register must have a parameterizable bit width, which is at least one bit. Since this register has a tri-stated

output, it can be put in parallel with other registers of this type, and thus a register file can be formed.

Registers without tri-statable outputs

These registers can store one value at a time, and it can be either read/write or read only. It can both be read and written in a clock cycle. The register must have a parameterizable bit width, which is at least one bit. Since this register does not have a tri-stated output, it can not be put in parallel with other registers, and thus this register can only be used when a Hyper design requires a register file with a single location.

Flip-Flop Registers

Flip-flops are single bit standard cell registers that are thought of as D flip-flops. These flip-flops are assumed to be rising edge triggered (if a flip-flop latches its data at the falling edge of the clock, this can be indicated in the library specification).

4. 2. 2 The Register Selection Process

The register selection is divided into two separate phases:

1. Determine the size of each Hyper register file, as annotated on the flowgraph.

2. Determine which register or combination of registers offers a better solution. This is done on a per-register-file basis. Use the registers that result in the smallest area.

During the first phase, the flowgraph is used to determine how many registers there are in each register file. During the second phase, the best (smallest) solution is determined for each register file. This is accomplished by considering which combinations of the available hardware offers a viable solution. The smallest viable solution is chosen as the optimal.

4.3 FLOWGRAPH TRANSLATION

The second step in the target independent phase is the flowgraph extraction step. During this step, the *Hardware Mapper* takes a scheduled flowgraph [Chu89][Rab91] and extracts the necessary information from it. Figure 4.5 shows a simple example of a flowgraph of a third order finite impulse response filter (FIR3).

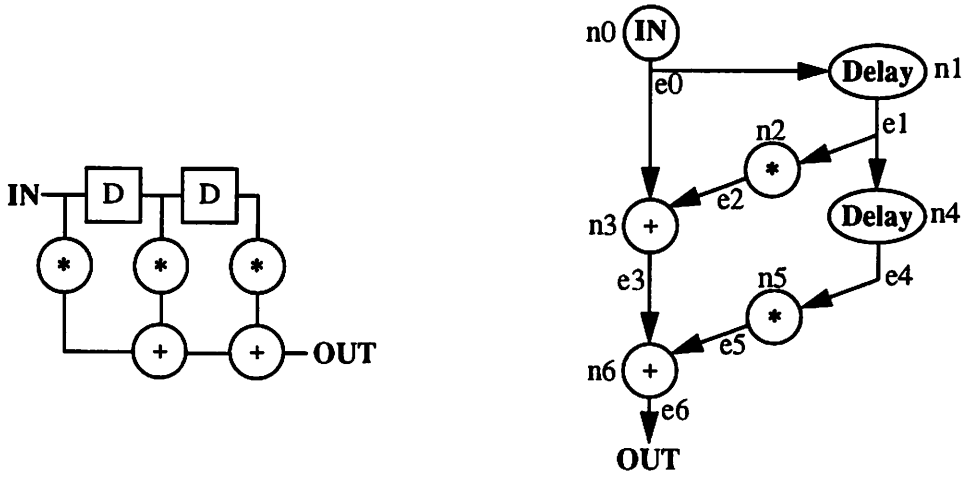

FIGURE 4.5 : FIR3 block diagram and flowgraph

The information which the *Mapper* extracts from the flowgraph is shown in Figure 4.6 and will be outlined in the following sections.

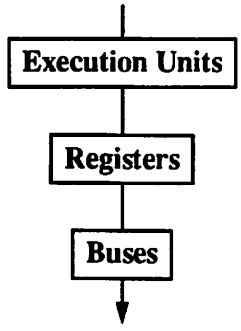

FIGURE 4.6 : Information which is extracted from a flowgraph

4.3.1 Execution Units

A scheduled flowgraph contains information about how, where and when an operation should be performed. The "how" is contained in the node attribute "exu_class". This gives the name of a hardware unit that can be found in the hardware library. The "where" is contained in the node attribute "exu_instance", and it specifies the name of an instance of the given hardware unit. The "when" is contained in the node attribute "control_step", and it specifies the clock cycle during which this operation will be performed on this instance of the hardware unit.

Example

The node n3 from Figure 4.4 is shown in all its detail below.

```
(NODE
    (NAME n3)
    (MASTER add)
    (ATTRIBUTES
        (exu_instance adder#0)
        (exu_class adder)
        (control_step 3)
    )
    (IN_EDGES (e0 e2))
    (OUT_EDGES (e3))
)
```

4.3.2 Registers

Registers are used to store intermediate results between operations. The information about registers is associated with edges (the data flow between operations). As for the execution units, the *Mapper* can find which hardware units to use and the names of all the registers that are needed for a design. There is no explicit information about when a register is written or read, but it can be inferred from the node which precedes a given edge.

Example

The edge "e3" from Figure 4.4 is shown in all its detail below. It is given that the data that flows between nodes n3 and n6 should be stored temporarily in a register called adder#0_reg0_1, which is an instance of the hardware unit tspcr (true-single-phase-clock-register). Furthermore, it can be inferred that adder#0_reg0_1 should be written in the cycle in which its input node (IN_NODE) n3 is operating, namely cycle 3 (control_step 3). It can also be inferred that the register should be read during the cycle in which its output node (OUT_NODE) n6 is operating, namely cycle 4.

```
(EDGE
    (NAME e3)
    (CLASS data)
    (ATTRIBUTES
        (storage_class tspcr)
        (storage_instance adder#0_reg0_1)
    )
    (IN_NODES (n3))
    (OUT_NODES (n6))
)
```

4.3.3 Buses

The nodes in a flowgraph contain information about which hardware units are used in a design. The connections between units is specified implicitly by the edges that connect nodes. The *Hardware Mapper* initially creates as many physical buses as there are unique point-to-point connections between hardware units. The *Mapper* looks at the connections that are represented by all the edges in a flowgraph, and finds the data types of connection, including when data is cast from one type to another. If a user so desires, the *Mapper* can subsequently reduce the number of buses by merging them together, creating time multiplexed buses. The merging of buses is covered in greater detail in the next section, "Bus Merging".

4.4 BUS MERGING

The *Hardware Mapper* is capable of minimizing the number of physical buses in a design. This feature is offered as an option, but it is highly recommended if implementation area is of any concern. The area of a bus-merged implementation is often only 40 - 60% of the area of one which did not use bus merging.

The bus merging routines are organized as shown in the flowchart below (Figure 4.7). Fanin and fanout merging are mutually exclusive, while global bus

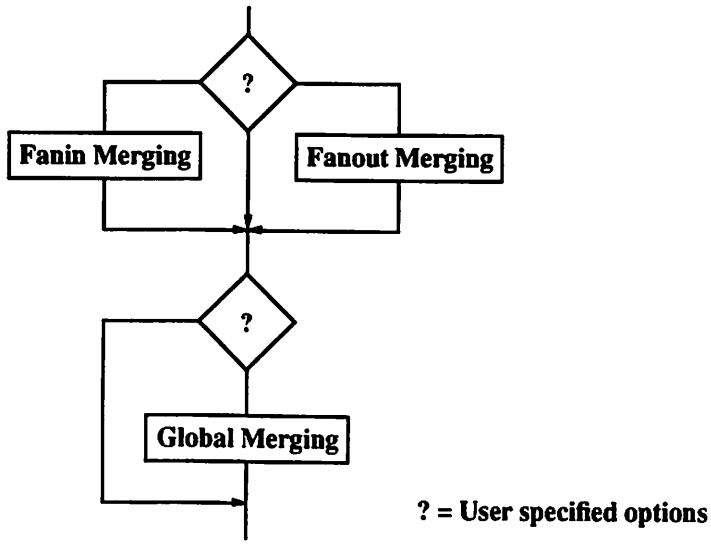

FIGURE 4.7 : Bus merging flowchart

merging can be combined with either of the fanin/fanout merging methods. The merging can also be bypassed (that is the default). The following three sections will describe each of the types of bus merging.

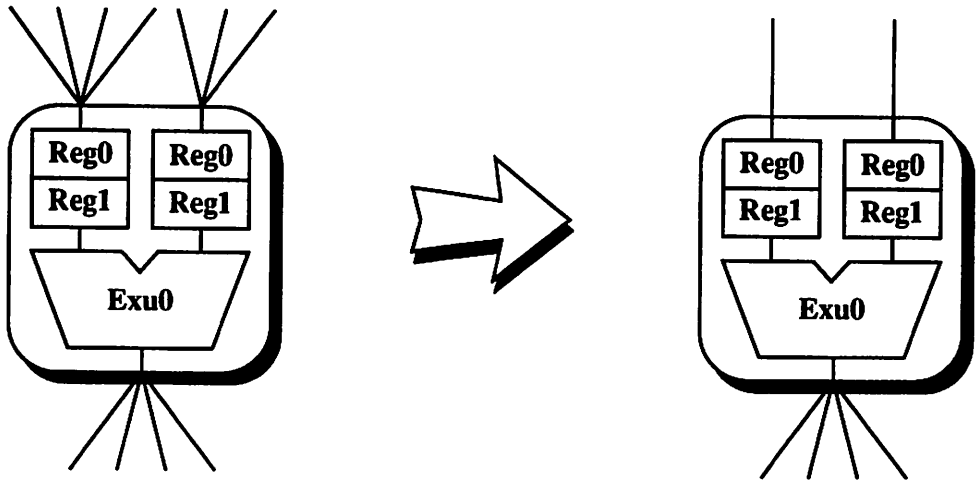

FIGURE 4.8 :Fanin bus merging.

4. 4. 1 Fanin Bus Merging

The fanin bus merging joins all the input buses of a register file into a single bus (See Figure 4.8). The philosophy behind this type of merging is that a register file can only write one value per cycle, and that value is presented by one of its fanin buses. Therefore, there is never more than one active input bus in a given clock cycle, and all the input buses can thus be merged.

The consequence of this type of merging is that there will be no need for multiplexers, while the need for tri-state buffers will increase.

4. 4. 2 Fanout Bus Merging

The fanout bus merging joins all the output buses of an execution unit into a single bus (See Figure 4.9). The philosophy behind this type of merging is that an execution unit can only generate one value per cycle, and that value is presented to all

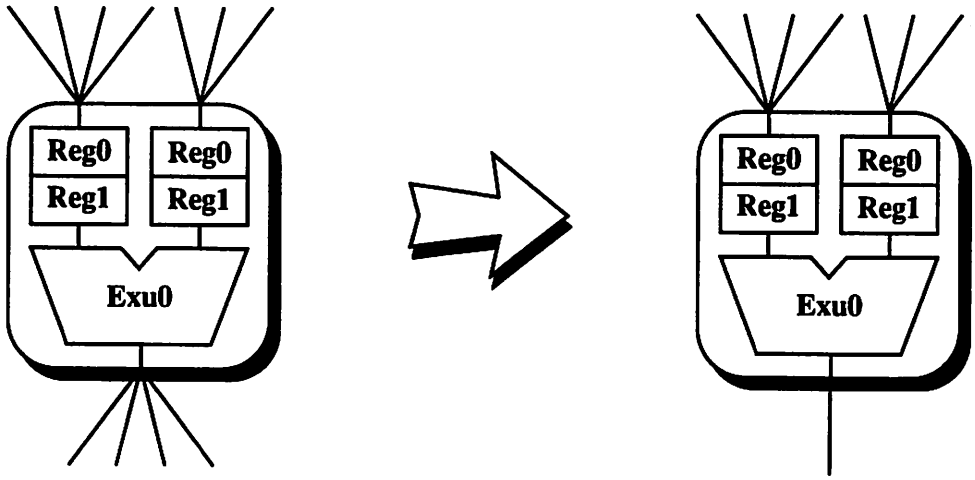

FIGURE 4.9 :Fanout bus merging.

of its fanout buses. Therefore, the buses always carry the same values in the same clock cycles, and all the output buses can thus be merged.

The consequence of this type of merging is that there will be no need for tri-state buffers, while the need for multiplexers will increase.

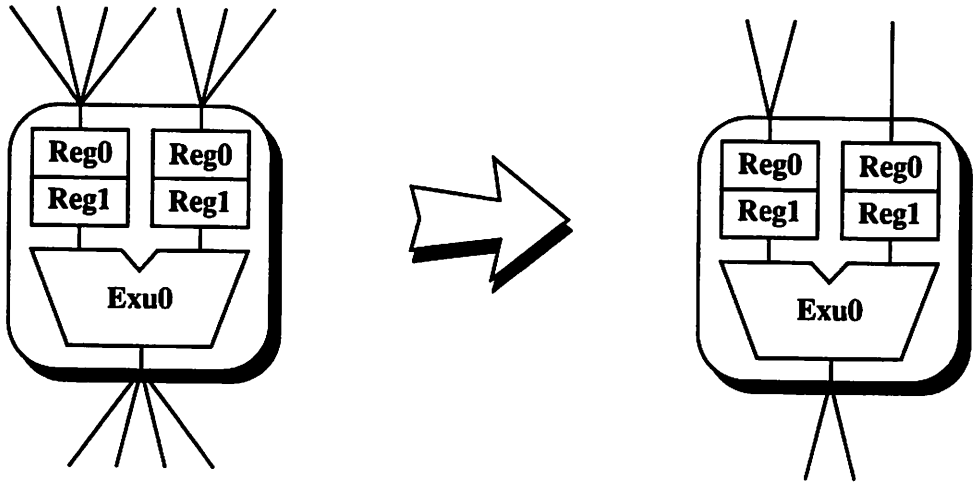

FIGURE 4.10 :Global bus merging.

4. 4. 3 Global Bus Merging

The global bus merging approaches the issue from a different angle. It considers all the global buses simultaneously and merges buses that are not active (transferring useful data from one unit to another) during the same cycles, such that the number of buses is minimized (See Figure 4.10). The merging procedure first determines which buses are incompatible (different widths, active in the same cycles etc.). It then performs a graph coloring to determine the minimal number of buses. Finally, it performs the actual merging.

Global bus merging normally achieves the best results (as measured in implementation area) out of the three kinds of merging. However, occasionally a combination of the three types of merging will yield the smallest area.

Example

A seventh order infinite impulse response filter (IIR7) was generated by Hyper. It has a critical path of 10 clock cycles and an available time of 16 clock cycles. There are 2 shifters, one adder and one subtractor in the design. Without any bus optimizations there are 21 buses in the design. The number of muxes and buffers are 8 and 25, respectively. The results of applying fanin, fanout and global bus merging are shown in the table.

	Global Buses	Muxes	Buffers	Rel. Area
No Optimizations	21	8	25	1.00
Merge Inputs	9	0	25	0.61
Merge Outputs	7	8	6	0.55
Global Merging	5	6	13	0.49
Merge Inputs and Global	8	0	25	0.58
Merge Outputs and Global	5	6	6	0.48

4.5 BUFFERS AND MULTIPLEXERS

After bus merging has been applied, buffers and multiplexers are added to the design. The buffers are primarily needed to drive the relatively large loads of global buses. However, another important feature of buffers is their capability of being turned on and off. This is an essential feature when there are many different buffers that need to drive a single bus. When all the buffers are turned off (tri-state), one buffer can be turned on and gain full control of the bus. The multiplexers have a different task. Instead of driving a single bus, they have to select one out of many buses. By selecting one bus at a time, a multiplexer can ensure that a register file only receives one value at a time. Thus, multiplexers play an important role in the implementation of a Hyper architecture.

There is no information available in a Hyper flowgraph concerning buffers and multiplexers. It is left to the *Hardware Mapper* to determine how many of these units are needed and which hardware units to use. The following two sections describe how the *Mapper* derives this information.

4. 5. 1 Buffer Selection

The buffer selection takes place after the global buses have been merged (if merging is requested by the user). At that point, it is known exactly how many buses there will be in the final implementation. It is also known which buses connect to each execution unit.

Buffers are always assumed to be placed at the output of execution units, and their function is to act as bus drivers. In cases when a bus is driven by different execution units during different cycles, it is necessary to use buffers that are tri-statable.

The buffer selection routine is very simple. It inserts a simple buffer between execution units and buses that are driven by only one execution unit. It inserts a tri-statable buffer between execution units and buses when a bus is driven by more than one execution unit. The *Mapper* does not consider the capacitive loading of buses when assigning buffers, but this could be taken into consideration to determine buffer sizes.

Example

A bus is driven by one execution unit. A simple buffer is inserted.

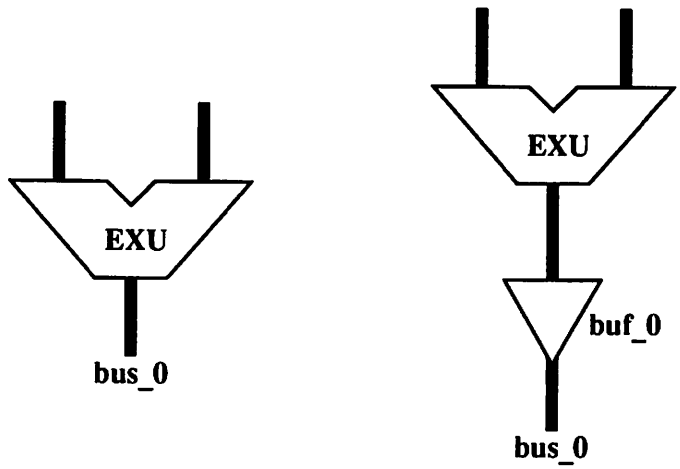

FIGURE 4.11 : Before and after buffer selection

Example

A bus is driven by two execution units. Tri-statable buffers are inserted at the output of each execution unit. Please see Figure 4.12.

4. 5. 2 Multiplexer Selection

The multiplexer selection takes place after the global buses have been merged (if merging is requested by the user). At that point, it is known exactly how many buses there will be in the final implementation. It is also known which register files the buses connect to. Multiplexers (muxes) are always assumed to be placed at the input of

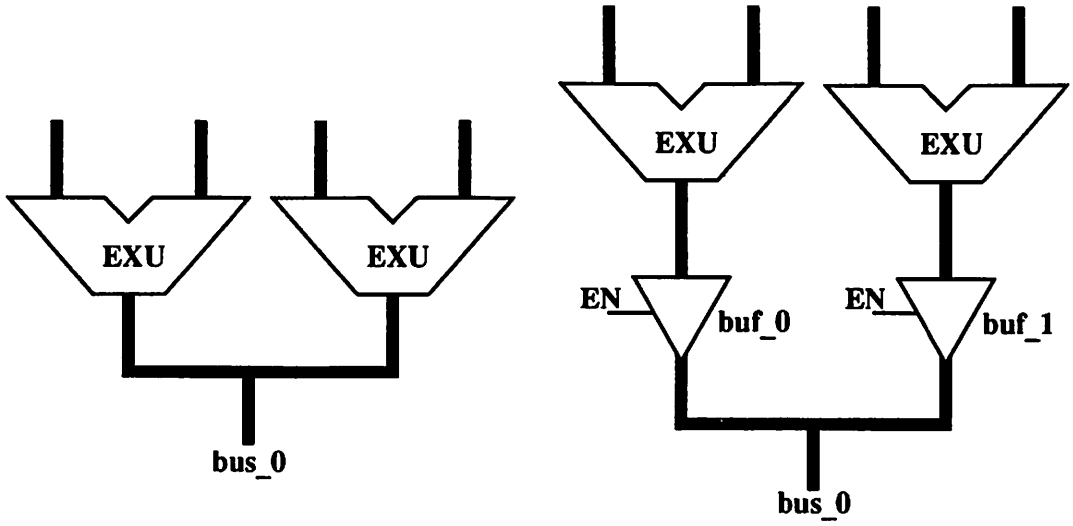

FIGURE 4.12 : Before and after buffer selection

register files. Thus a mux serves as a bus selector for the register file with which it is associated.

The multiplexer selection routines recognize first of all how many buses are connected to the input of a register file. Then it considers which of the available muxes is best suited.

Example

A register file has one bus connected to its input. The mux selection will not assign a mux, since there is no bus selection to be done.

Example

A register file has two buses connected to its input. The mux selection will assign a two-to-one (2:1) mux. Please see Figure 4.13.

Example

A register file has three buses connected to its input. The mux selection will try to assign a mux which has three inputs (a 3:1 mux). If a 3:1 mux is not available in the hardware library, the mux selection will construct a "mux-tree" using smaller muxes, in

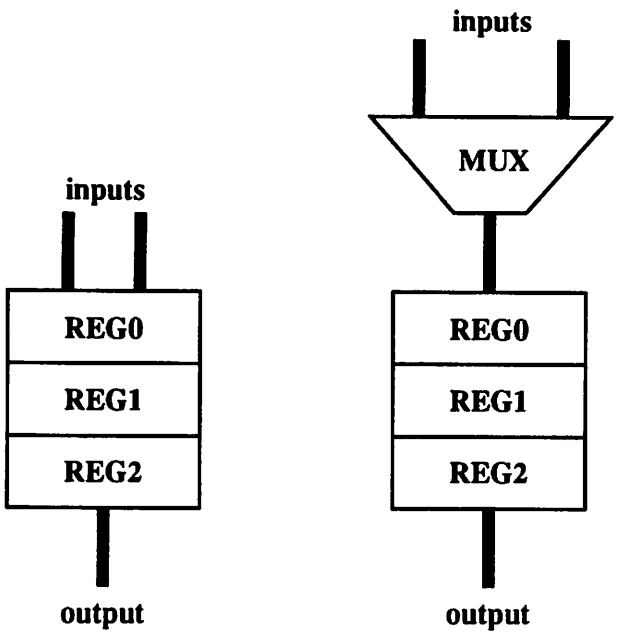

FIGURE 4.13 : Before and after mux selection

this case 2:1 muxes. The two possible mux selections (a 3:1 mux or two 2:1 muxes in a mux-tree) are shown in Figure 4.14.

4.6 CONTROL GENERATION

The last step in the target independent phase is control generation. "Control" in this context encompasses numerous different areas that all somehow are grouped under this designation. For example, there is global control, which involves synchronization of the chip as a whole. There is also control signals, which are very specific to individual hardware units, but have to be coordinated with the global control.

The control generation is partitioned into 4 steps (see Figure 4.15). The ordering of the first three steps is important as each step depends on the previous step. Section 4.6.1 explains how the state transition graph is derived. Section 4.6.2 will outline the control signal initialization. Section 4.6.3 covers the control signal assignment issues. Section 4.6.4 deals with control tables, while section 4.6.5 discusses

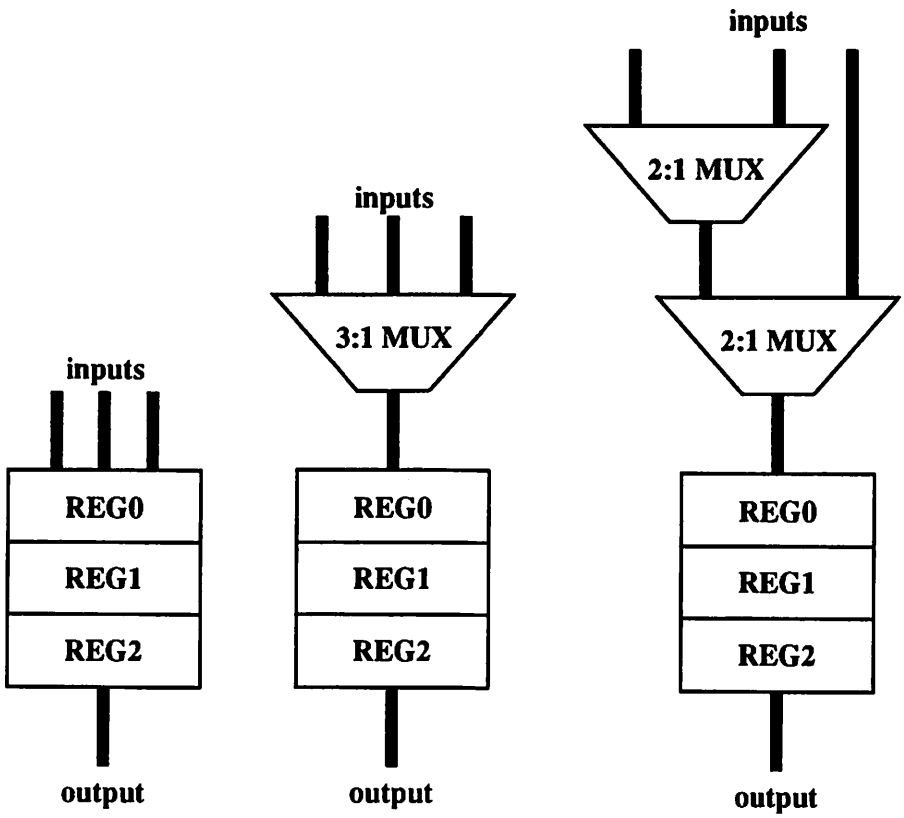

FIGURE 4.14 : Two different mux selections

signals that are used for synchronization with external circuits. Section 4.6.6 covers the whole area of data generated control signals.

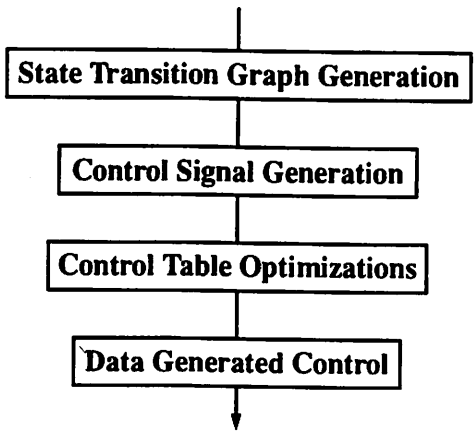

FIGURE 4.15 Flowchart of the control generation routines.

4. 6. 1 State Transition Graph

The *Mapper* generates a state transition graph based on the information available in the flowgraph. It keeps track of all the states and the operations that must take place during those states. It also determines which control signals will be used to determine control flow (like branches, jumps etc.). Currently, we only support two types of control flow: sequential and simple loops. Sequential flow is the simplest form. It does not require any control signals to flow from the datapath to the controller. In contrast, simple loops require that the controller can receive feedback from the datapath to determine when or if to branch.

Since state transition graph generation is not the major focus of the *Mapper*, one example will suffice to illustrate the methodology used.

Example

Figure 4.16 shows a small flowgraph. Each of the nodes have been scheduled to take place during a certain control step. The state transition graph that will be generated for this example is also shown in the figure. Each state has a list of the operations that must be performed during that state. Due to the assumption made in

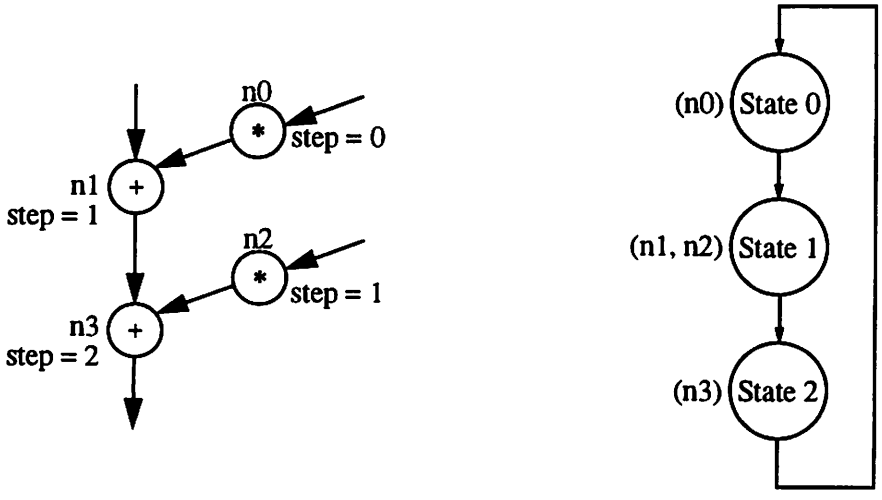

FIGURE 4.16 : Flowgraph and state transition graph.

Silage that all algorithms are recursive, the last state (State 2) is connected back to the first state (State 0).

4. 6. 2 Control Signal Initialization

The *Mapper* performs three simple steps when generating the control signals.

1. The hardware library is prompted for the names of all the control terminals of a certain hardware unit.

2. The *Mapper* creates a new control signal and assigns to it a default value. The default value can be either of these three:

   - 0 used for signals that should be zero when not in use (e.g output-enable signals on tri-stateable buffers)

   - 1 used for signals that should be one when not in use (e.g the $\overline{\text{write}}$ signal on a register)

   - X ("don't care") used for signals that can be either zero or one when the unit is not in use (e.g. multiplexer select signals)

3. In some cases the *Mapper* annotates the newly created control signal with information about which values to take under different circumstances.

   - Control Signals for Muxes: It is determined which values a control signal must take on to make the mux select input zero, one, two...

   - Control Signals for Shifters: It is determined how a control signal should be asserted when the unit has to perform a given function (right shift or left shift) and a certain shift amount.

- Control Signals of Other Functional Units: It is determined which values a control signal should take on when the functional unit performs a given function (e.g. the Carry-in of an adder should be zero when the unit performs the function "+").

Later in the generation process, the control signals are assigned values depending on when operations in the flowgraph are scheduled. The next section will outline this process.

Example

A functional unit capable of performing both additions and subtractions must have its carry-in signal set to zero when adding and one when subtracting. The *Mapper* creates a control signal, associates it with the functional unit, and annotates that it should be "0" for the function "+" and "1" for the function "-". The *Mapper* also sets the default value of the control signal to "X" (don't care), because it does not matter what the value it when the unit is not used. Later, during the control signal assignment phase, the control signal will be set to zero when required to do additions, one for subtractions and "X" when not in use.

4. 6. 3 Control Signal Assignment

All the control signals that are initialized in the step described above have only been assigned default values. A default value is essentially an "idle" value, meaning that a hardware unit remains idle as long as its control signals have default values. If the design is going to work properly, all units have to be "programmed" to perform their designated function at the right time. It is during the control signal assignment phase that all the hardware units in a design are "programmed" to do the right thing.

A scheduled flowgraph contains all the necessary information about when operations must take place. The *Hardware Mapper*'s job is to extract this information and assign proper values to the control signals that "program" the behavior of the chip.

The assignment process is conceptually simple. The *Mapper* goes through all the operations (nodes) in a flowgraph one by one. For each operation, there are a number of hardware units that have to be "programmed" or enabled. First, the registers that hold the source operands of the operation have to be accessed. Secondly, the functional unit that performs the operation may need to be programmed to do so. Third, the buffers that drive the result of the operation onto a bus, and the multiplexers that select that bus have to be enabled to do that. Finally, the registers that are going to store the result have to be readied for writing. These five types of units are shown in Figure 4.17. Figure 4.18 illustrates how the flow of data passes through the same five types of units.

Source Register Access	The source registers for the operation are set for reading. If the register file has more than one location (i.e. when there is more than one register in the file) all the other registers are set to $\overline{\text{read}}$ (don't read).
Functional Units	The functional unit is set to perform the function given by the operation.
Buffers	The buffers which drive data across global buses are enabled.
Multiplexers	The multiplexers at the input to the destination register file is set to select the proper input bus
Destination Register Access	The destination registers are set up for writing.

FIGURE 4.17 : Control signal assignment follows the flow of data.

When a control signal is assigned a value it is inserted in a control table. A control table can either hold all the control signals in a design or just the signals from a group of units. The control tables will be discussed in greater detail in the following section.

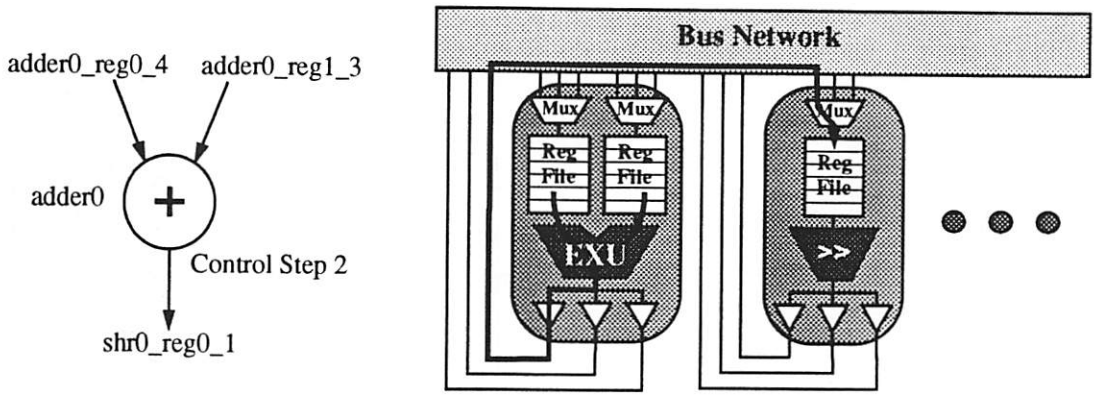

FIGURE 4.18 : Control signal assignment is analogous to the flow of data.

4. 6. 4 Control Tables

A control table is a medium the *Mapper* uses to keep track of control signal assignments as well as control signal partitioning. It may represent just the control signals needed for one group of units, or it may represent all the control signals in the chip. A control table is essentially a collection of control signals and their values. It is a two dimensional structure, with time in the vertical direction (measured in clock cycles or states) and control signals in the horizontal direction. The next two sections outline the procedures for initializing and optimizing control tables.

4. 6. 4. 1 Control Table Initialization

When a control signal is assigned a value, an entry is made in the control table to which the signal "belongs". Whether the table is global or local to a cluster is not an issue at this stage. The table has one row for each control signal and one column for each state in the state transition graph.

Example

Figure 4.19 shows a control table.

State	src_reg0_0_READ	src_reg0_1_READ	src_reg1_0_READ	adder0_CARRYIN	buffer0_ENABLE	mux0_SELECT	dest_reg0_0_WRITE
0	0	0	0	X	0	X	0
1	1	0	1	0	1	0	1
2	0	0	0	X	0	1	0
3	0	1	1	0	1	0	1

FIGURE 4.19 : Sample Control Table.

When src_reg0_0 is being read, all other registers in the same register file are disabled. The adder performs the function + in states 1 and 3. The carry-in signal is therefore "0" in those states. The adder is not used in states 0 and 2, and consequently the carry in is set to "don't care" ("X").

4. 6. 4. 2 Control Table Optimization

After a control table has been initialized, it contains quite some inefficiencies. The *Mapper* will optimize a control table to eliminate as much redundancy as possible. The optimization contains five phases that are ordered such as to not disable the following steps. The five phases are:

1. Eliminate constant control signals from the table. This includes signals with don't cares that do not conflict with the assigned values.

2. Eliminate duplicate (identical) signals from the table.

3. Eliminate signals that are the inverse of each other.

4. Eliminate signals that only differ in places where one of them has a don't care.

5. Don't care elimination. This is an option which can be requested when running the *Mapper* with the flag -x. All don't cares remaining after the above optimizations are set to the last non-don't care value for that signal. This may help improve the power consumption of the control logic.

Example

This example shows the effects of control table optimization. Each column has a column number and a control signal associated with it. Each row represents a state.

State \ Column	0	1	2	3	4	5
0	0	0	1	0	0	X
1	X	1	0	1	X	0
2	0	0	1	0	X	0
3	X	1	0	1	1	1
Control Signals	A	B	C	D	E	F

By performing the control table optimizations (except the don't care resolution) the table is reduced to this:

State \ Column	1	5
0	0	X
1	1	0
2	0	0
3	1	1
Control Signals	B,inv(C),D,E	F

Signal A was removed by constant elimination. Signal C was recognized to be the inverse of B. Signal D is identical to B. Signal E does not have any conflicts with B and can thus be merged with B.

After don't care resolution (which only is performed upon request) the resulting table is:

State	1	5
0	0	1
1	1	0
2	0	0
3	1	1
Control Signals	B,inv(C),D,E	F

Column 5 had an "X" in state 0. It was resolved to a "1" because the "previous non-don't care value" is "1" in state 3. Notice that "previous" can span across two different sample periods. In this case column 5 received a "1" since the preceding state (State 3 from the previous sample period) was "1".

4. 6. 5 Input and Output Valid Flags

Beyond the necessary control signals, the *Mapper* also generates signals that aid the synchronization with external circuits. There are as many of these input and output valid flags as there are algorithmic inputs and outputs (one flag per input/output in the Silage algorithm). These flags indicate when an input sample *should* be valid or when an output sample *is* valid. They are essential in any design for synchronization.

Example

A chip has one ioUnit but two "algorithmic" inputs. This can happen when the Silage description of a design has two inputs, while the allocation phase of the design only allotted one (time-shared) ioUnit. The input valid flags dictate when each of the two signals must be presented to the chip. Thus, a designer can easily determine, e.g. when to switch a mux, or when to "pop" a FIFO.

Example

A chip has one output bus and one "algorithmic" output. It is clear that the output will be presented on the output bus, but when? Using the output valid flag, it is easy to determine when to latch the output into a FIFO or when to activate another chip which may be waiting for the output.

4. 6. 6 Control Paths

A control path is defined loosely as a network of data-generated signals which control parts of the datapath. A simple example of a control path is the signal "SEL" in Figure 4.20. SEL is generated by comparing two data, A and B, and it decides (controls) whether the data C should take on the value of A or B.

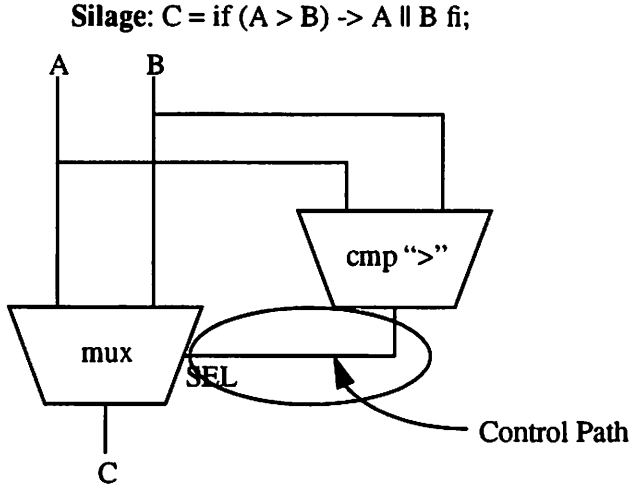

FIGURE 4.20 : Example of a Control path

Example

Control paths can be arbitrarily complex, since Silage allows arbitrarily complex arguments in an *if* statement. Figure 4.21 shows the implementation of a piece of Silage code.

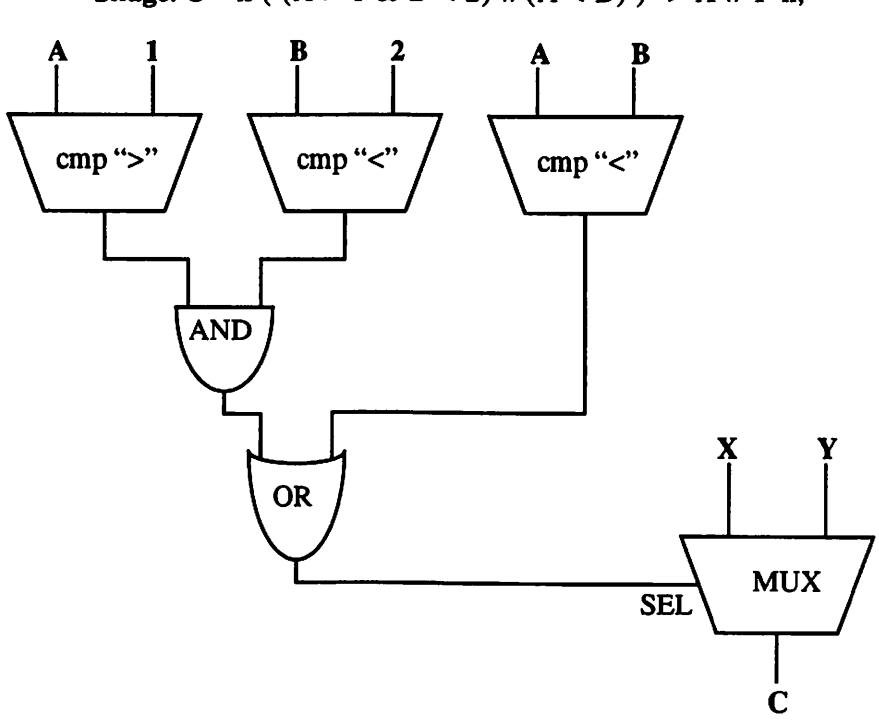

FIGURE 4.21 : Example of a complex control path

The *Mapper* scans the flowgraph for edges that belong to the boolean class (CLASS boolean). These edges will normally originate from comparators (or boolean nodes), and end at multiplexers (or boolean nodes). The flowgraph does not contain information about where or how to store intermediate control path signals, so the *Mapper* must derive this information. This derivation is based on the assumption that all inputs to a control path (or a boolean network) should be stored as intermediate results. In the example above, the *Mapper* would decide to assign single bit registers to each of the outputs of the comparators. When scanning the boolean edges, the *Mapper* also keeps track of the boolean functions applied to each signal. All the boolean functions are mapped to standard cell blocks found in the library index.

5

VHDL GENERATION

Hardware mapping was defined in the introduction as the process that interfaces high level synthesis tools with layout generation tools. Thus, on the input side of the *Hardware Mapper* is a realm of conceptual design, where abstractions are used to model actual hardware implementations. On the output side, the *Mapper* must deliver very specific, completely specified design descriptions.

The format in which a design is described is called a "hardware description language". VHDL [VHD87] is one such language. It was standardized by the IEEE in 1987. Since then, VHDL has gained widespread acceptance among digital hardware designers. The language is also gaining commercial acceptance, and there are various products available for simulation or layout generation.

The goal of the *Hardware Mapper*'s VHDL generation is to translate the target independent information (explained in chapter 4) to a VHDL description. This chapter outlines the basic methodology used. Section 5.1 describes the concept of VHDL headers. Section 5.2 explains the VHDL file hierarchy. Section 5.3 shows the way casts are handled, and section 5.4 discusses the convenient VHDL simulation interface. (Please see Figure 5.1).

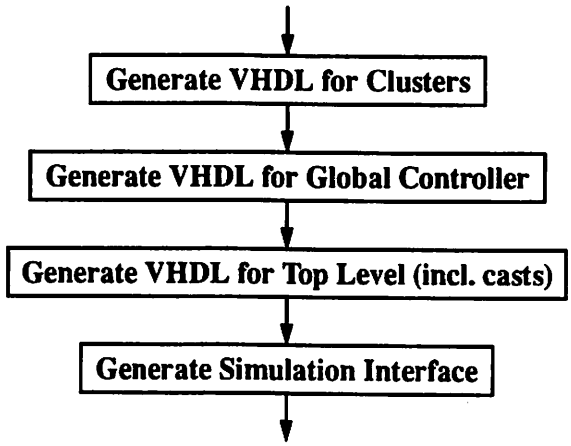

FIGURE 5.1 : VHDL Generation Flow

5.1 VHDL Headers

One requirement posed by VHDL is that all libraries must be declared before they are referenced. That may not appear to be a problem, but it is impossible for the mapper to derive this information.

Seen from the *Hardware Mapper*'s perspective, there are two kinds of libraries: The libraries specified by the designer (containing the VHDL models the designer wants to use), and the libraries that are required by VHDL (typically describing data types and defining basic operations). While the designer explicitly names the first library, the *Hardware Mapper* has no way of knowing which libraries define the basic data types and operations in VHDL. In order to allow the designer to provide this information, the concept of a VHDL header file is introduced.

At the top of each VHDL file generated by the *Mapper* a header will be inserted. The header is taken directly from a file called "vhdl.header" which must be placed in the path given for the library specified on the command line. In most cases, one VHDL header will suffice for all designs using one particular hardware library.

Thus, the VHDL header file will usually only have to be written once for each hardware library.

Example

The following is an example of a header file used for the VHDL models of Lager's dpp library.

```
LIBRARY dpp; -- The library of cells
LIBRARY ieee; -- The library which defines basic types and operations
USE ieee.logic.all; -- Definition of basic types and boolean operations
USE ieee.arithmetic.all; -- Definition of basic arithmetic operations
USE std.textio.all; -- Definition of basic text capabilities
```

5.2 The VHDL File Hierarchy

The VHDL file hierarchy consists of three levels, two of which are generated by the *Hardware Mapper*. The highest level is called datapath.vhd. Its inputs and outputs correspond directly to the actual chip I/Os. In cases where I/Os have been merged together there will be fewer I/Os in the chip than expected from the Silage description. The file datapath.vhd also has a number of "I/O valid signals". There is one such signal for each input or output in the Silage description. Please see section 4.6.5 for details about the input/output valid signals.

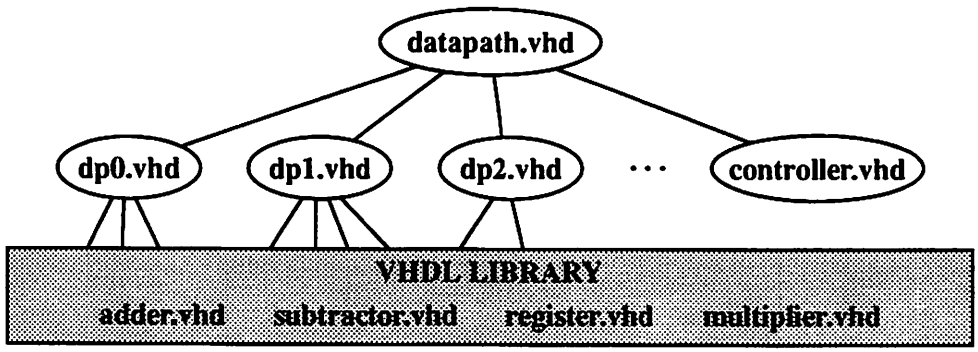

FIGURE 5.2 : The VHDL File Hierarchy.

The second level of hierarchy consists of multiple files. Their names are dp0.vhd, dp1.vhd... and there are as many of them as is necessary for the particular design. Each of these files contain one datapath cluster. A cluster may contain any number of multiplexers, registers/register files, functional units or buffers.

The second level also includes the global controller description, controller.vhd. The controller is written in simple procedural VHDL, with two case statements, one to determine the next state based on the current state and the feedback from the datapath, and another to assign the control signals based on the current state.

The third level of the hierarchy is not generated by the *Mapper*. This is the level at which the library models reside. These models are kept in libraries for maximum reusability and minimum maintenance, and they often correspond directly to actual hardware units.

5.3 Casts

Silage [Hil85], the input language of Hyper, has a construct called a "cast". It allows a designer to convert from one data type to another or to force a specific data type on a variable. It is similar to the casts used in the C programming language, but Silage adds the capability of specifying the number of bits used to represent a variable. Figure 5.3 shows examples of casts in both C and Silage.

The implementation of casts is a very complicated task. We have chosen to implement casts in a static fashion, i.e. casts are hard-wired. This strategy is flexible enough to handle very complicated cases without needing additional hardware (like shifters and multiplexers) to align bits properly. In some cases, however, there will be a greater number of buses needed in a design.

Casts in C

```
unsigned u;
char c;
int i;

i = (int) c;
...
u = (unsigned) i;
...
```

Casts in Silage

```
a = fix<6,3> (1.0);
b = unsigned<8,4> (a);
```

FIGURE 5.3 : Cast examples

When there are casts in a design, they are handled at the highest level of the hierarchy, i.e. the in the datapath.vhd file. The philosophy is to not worry about bit manipulations inside a cluster, but rather take care of all of them in one place. Thus, all signals in a cluster have the same bit widths as the hardware units that they are connected to. This includes the inputs and outputs of a cluster.

There are a few terms in the *Mapper*'s cast handling that will be explained here. Figures 5.4. and 5.5 illustrate these terms.

1. Bit Selection: A number of bits from an output are selected and connected to a bus.
2. Zero Padding: A number of bits at the LSB end of an input are connected to a logic '0'.
3. Sign Extension: A number of bits at the MSB end of an input are connected to a the sign bit of a bus that is also connected to this input.
4. Zero Extension: A number of bits at the MSB end of an input are connected to a logic '0'.

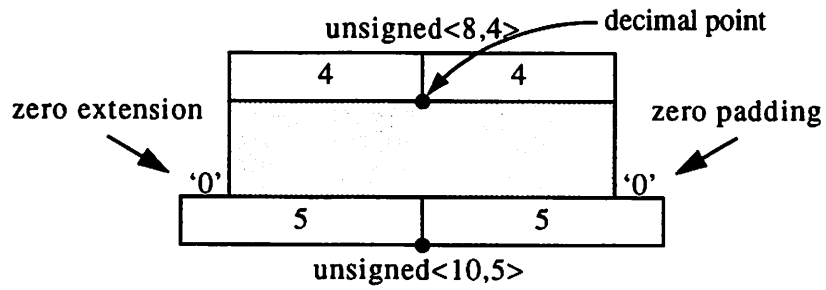

FIGURE 5.4 : Casting from unsigned<8,4> to unsigned<10,5> shows zero padding and zero extension.

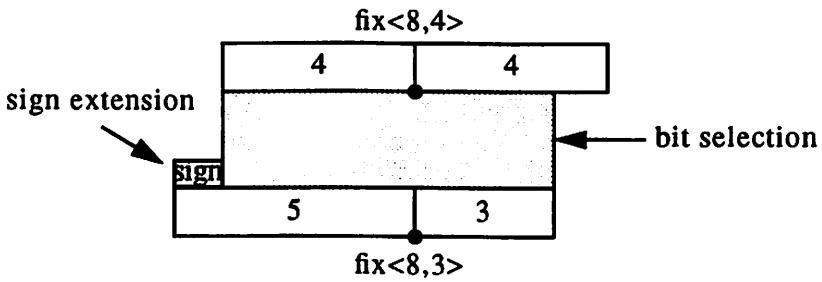

FIGURE 5.5 : Casting from fix<8,4> to fix<8,3> shows bit selection and sign extension.

To handle some obscure cases the following terminology was introduced. Figure 5.6 illustrates these concepts.

1. Width: The number of bits of a bus.
2. Terminal Base: The bit number of the lowest order bit on a terminal that is connected to a bus.
3. Net Base: The bit number of the lowest order bit on a bus that is connected to a terminal.

Example

This example shows a case where two different casts are used between an adder and a subtracter. The silage code of such a case is shown below, and the flowgraph is shown in Figure 5.7.

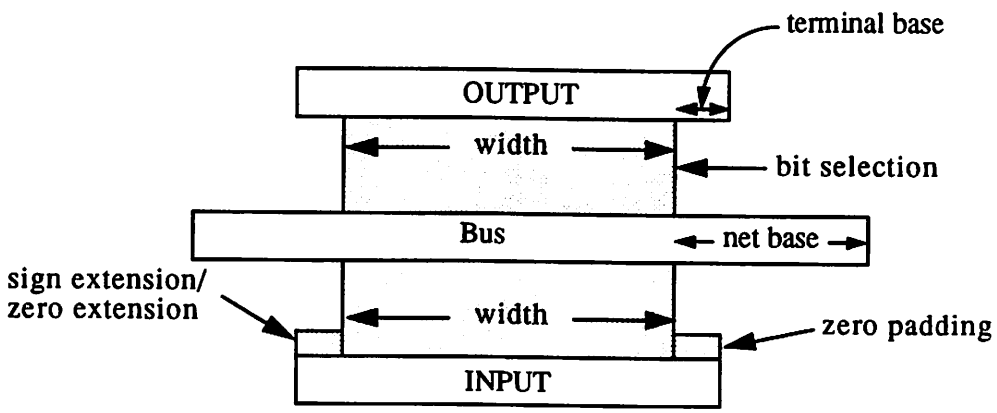

FIGURE 5.6 : Illustration of width, terminal base and net base.

```
Silage:
/* A is fix<8,7> */
/* E is fix<8,1> */
B = A + 0.25;
C = fix<8,1>(B);
D = C - 1.0;
F = E + 2.0;
G = fix<8,7>(F);
H = G - 0.25;
```

A and B have the same data type, fix<8,7>. The value of B is assigned to C through a cast which changes the data type to fix<8,1>. Similarly, E and F have the data type fix<8,1>, and the value of F is assigned to G through a data type conversion to make it fix<8,7>. There are two 8 bit additions, and two 8 bit subtractions. If the additions and subtractions all are scheduled on different units, we have the situation depicted in Figure 5.8. If the two addition nodes are scheduled on the same adder but the two subtractions are scheduled on two different subtractors, then we have the case shown in Figure 5.9. Finally, if the additions are performed on one adder, and the

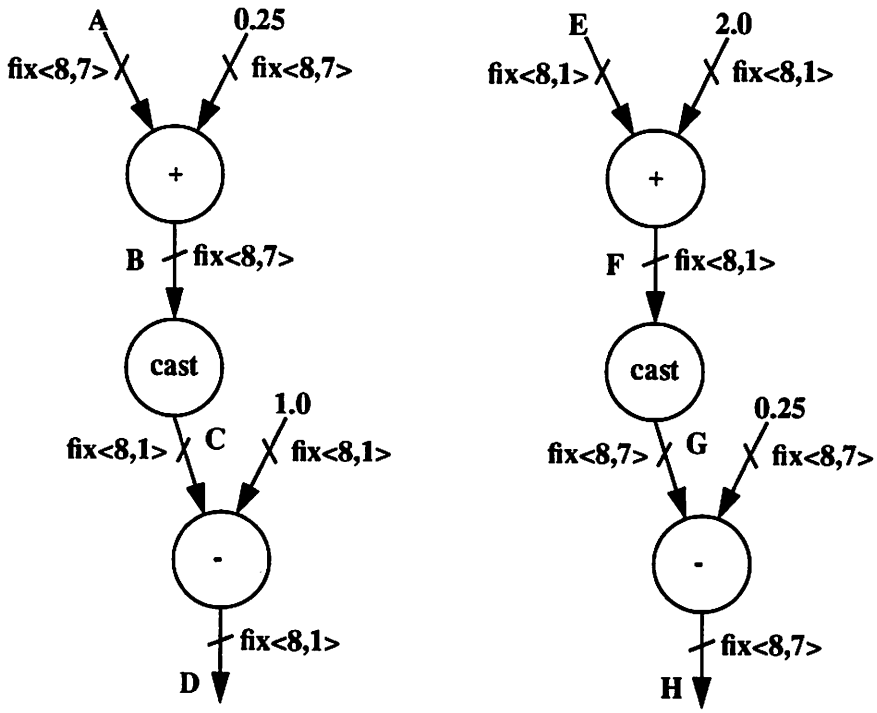

FIGURE 5.7 : Flowgraph for the Silage code in the example above.

subtractions on one subtractor, then we have the case shown in Figure 5.10. This "cross" transfer will be implemented as shown in Figure 5.11.

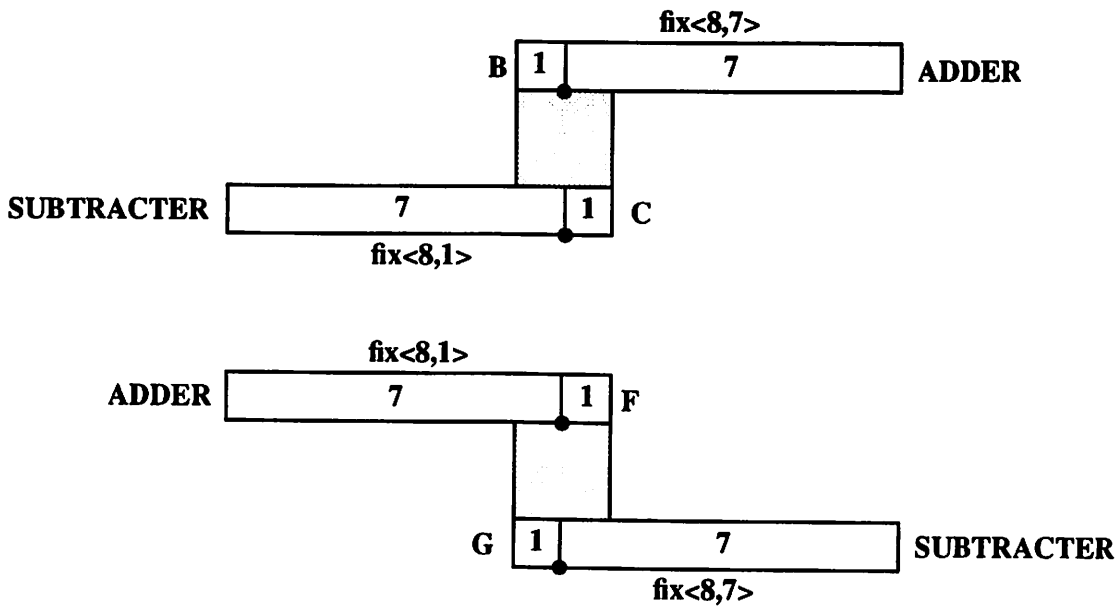

FIGURE 5.8 : Two adders and two subtractors.

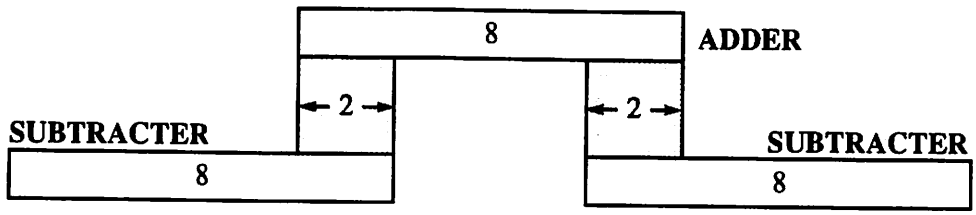

FIGURE 5.9 : One adder, two subtracters.

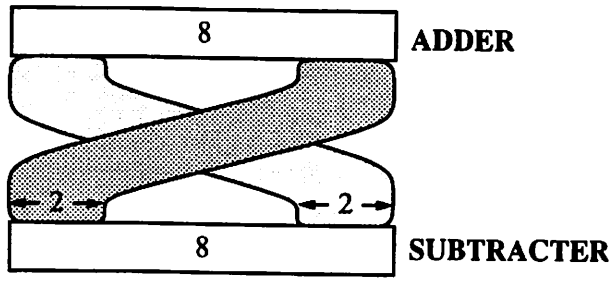

FIGURE 5.10 : One adder, one subtracter.

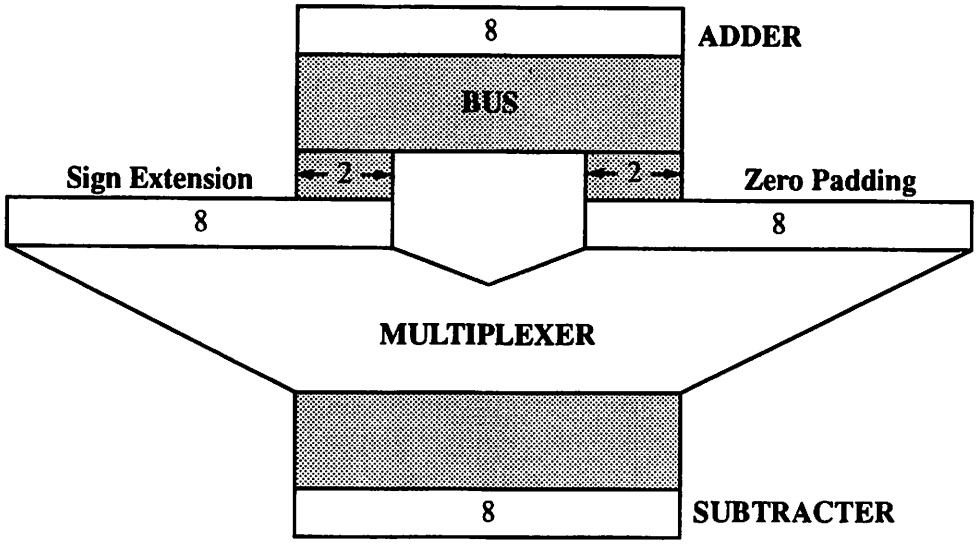

FIGURE 5.11 : Implementation using a multiplexer.

The implementation of the situation shown in figure 5.11 is to connect the full width of the adder's output to the bus. Temporary signals have to be declared with the width of the input of the subtractor. These signals are assigned the proper bits from the bus, and then sign extended or zero padded as shown in the figure.

5.4 Simulation Interface

Besides generating all the VHDL files representing a design, the *Hardware Mapper* adds a convenient simulation interface which automates VHDL simulation. The interface relies on three VHDL units. A "generator" converts simulation stimuli from an input file to VHDL signals. A "logger" converts simulation results to numbers and stores them in a file (in xgraph format). A "clock generator" (clkgen) generates the initial chip reset as well as the clocks.

In addition to generating a "wrapper" layer for simulation purposes, the Mapper generates a Makefile which completely automates the compilation and simulation of a design.

Once the *Hardware Mapper* completes, a Synopsys VHDL simulation can be done simply by going into the vhdl directory, and typing "make" (the only prerequisite is that the path and the environment variables must be set for the simulator, and a library mapping file must be present (for Synopsys it is $HOME/vhdl.uof)). Once the make completes (after compilation and simulation), there should be as many log files as there are outputs in the Silage description (if there are two outputs, A and B, then there will be two files called A.xg and B.xg)

6

SDL GENERATION

The acronym *SDL* stands for Structural Description Language [Bro92]. It is a netlist language that allows structural description of designs, including designs that are composed hierarchically. SDL is the input language for the Lager layout synthesis tools [Lager91]. Lager, under the control of an automated "design manager" (DMoct [Lager91]), can generate a layout from an SDL description of a design.

SDL was chosen as one of the output languages supported by the *Hardware Mapper* to provide a complete flow from algorithm to layout, using only public domain software. Hyper performs the synthesis from algorithm to structural representation, and Lager completes the path by generating layouts.

The *Mapper*'s SDL generation consists of two separate phases (see Figure 6.1). First, the target independent data structure is transformed such that it can be directly mapped to SDL without violating any of the rules of that language. Also, the design may be conditioned in an attempt to help Lager produce good, area efficient layouts. Secondly, the transformed design is mapped to the SDL format.

This chapter first explains the transformations that are necessary, and some that are desirable, before mapping to SDL. The mapping process is then described, including sections on the file hierarchy, the controllers and cast issues.

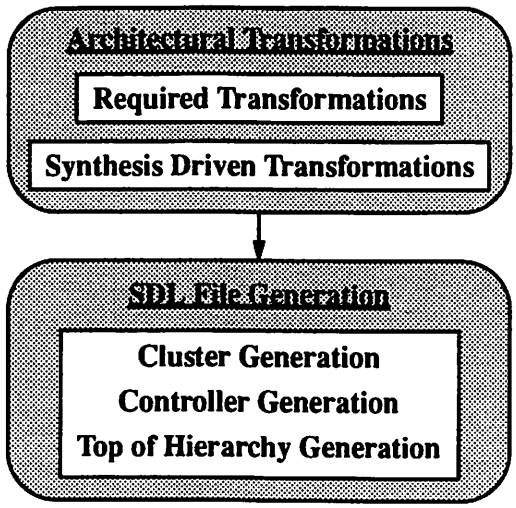

FIGURE 6.1 : SDL Generation Flow

6.1 Required Transformations

There are two types of design transformations that must be completed before a design can be successfully mapped to SDL. The first is to separate components by type and the second is to separate components by width. These two transformations will be described in detail below.

6.1.1 Separation by Type

The first transformation that must be performed before generating SDL is to separate array-type components from datapath-type components. This transformation is required because SDL handles the two types in very different ways, and thus is not able to mix components of the different types in one file.

In SDL an "array-type" component is normally one which is tileable in two dimensions, typically the bit width and some other dimension. Examples of array components are multipliers (tileable in the bit widths of the two inputs) and memories (tileable in the bit width and the number of words). A "datapath-type" component is one

which is tileable in one dimension, namely the bit width. Examples of datapath components are adders, shifters, registers.

The hardware library is capable of indicating the type of a component. This ability is derived from the fact that all units must be presented to the library in either of the files rb-dp, rb-array or rb-std. If a component is found in rb-dp it is considered to be a "datapath-type", and it can only be grouped with other datapath components in an SDL file. Likewise, if a component is found in rb-array it is considered to be a "array-type". For array types, however, the mapper never groups array components together. Normally, array components are significantly larger and/or have irregular dimensions. By keeping each array component by itself, the final (manual) placement will typically yield smaller layouts than if grouping is performed.

6.1.2 Separation By Width

Once the arrays and the datapaths have been separated from each other, the *Mapper* has to separate datapath components that have different bit widths. This transformation is required by dpp (the Lager tool which handles datapath components). After this separation is complete, all hardware units within a cluster will have the same bit widths.

6.2 Synthesis Driven Floorplanning

Beyond the required transformations, the *Mapper* can perform some transformations which can assist Lager in achieving respectable layouts. These transformations are closely connected with the floorplanning issues, and I will therefore consider the two together.

There is an automatic and a manual mode of operation. In the automatic mode the *Mapper* performs the cluster grouping and floorplan generation according to its

understanding of a Hyper floorplan model. This is described in detail in section 6.2.2. In the manual mode, a file must be supplied which guides the *Mapper* in which units to group (if any), and the relative ordering of units in the floorplan. The manual mode is described in detail in section 6.2.3.

6. 2. 1 Hyper Floorplan Model

The underlying assumption in the merging and floorplanning routines is the floorplan model shown in Figure 6.2. This model may have any number of units, and it can be expanded arbitrarily to the right. Each of the clusters in the model may contain any number of execution units, multiplexers, registers, and buffers. In addition to the clusters, there may also be any number of control blocks. The idea behind this plan is to have one central communication channel in which all global buses are routed from source to destination. The clusters of hardware units are placed in one row below and one row above, with input and output connections facing the central channel. Each cluster may or may not have its own control block, or two clusters may share a control block which lies between them.

Figure 6.2 illustrates some of the combinations mentioned. Cluster0 has no control block next to it. Cluster1 and Cluster2 each have their own control blocks, but they are both located between the clusters. Cluster3 and Cluster4 share one controller and Cluster5 has its own controller.

6. 2. 1. 1 Motivation

The motivation behind the cluster merging comes from the fact that most clusters are differently shaped. Figure 6.2 shows all clusters as being equally sized, but that is rarely the case. A more typical case is shown in Figure 6.3. There is one (or a few) units that are tall, and the rest of the units have various heights. In pursuit of area efficient layouts, we can merge the small units, thus obtaining an architecture with

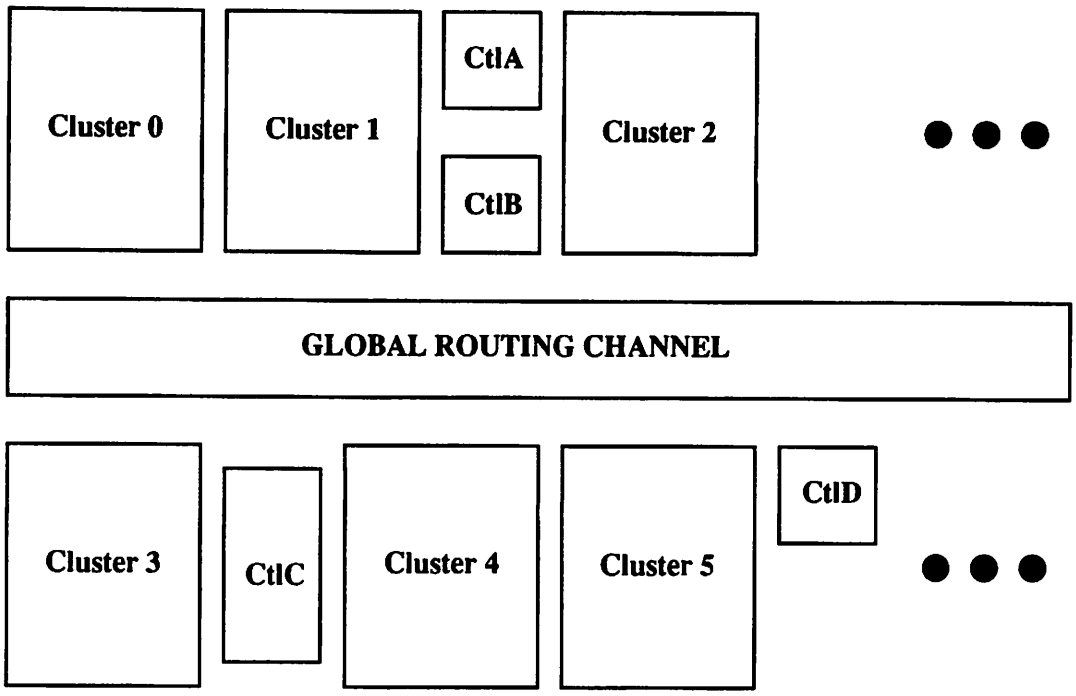

FIGURE 6.2 : Generic Hyper Floorplan

fewer units that are more evenly shaped. The result of merging some of the small units from Figure 6.3 is shown in Figure 6.4.

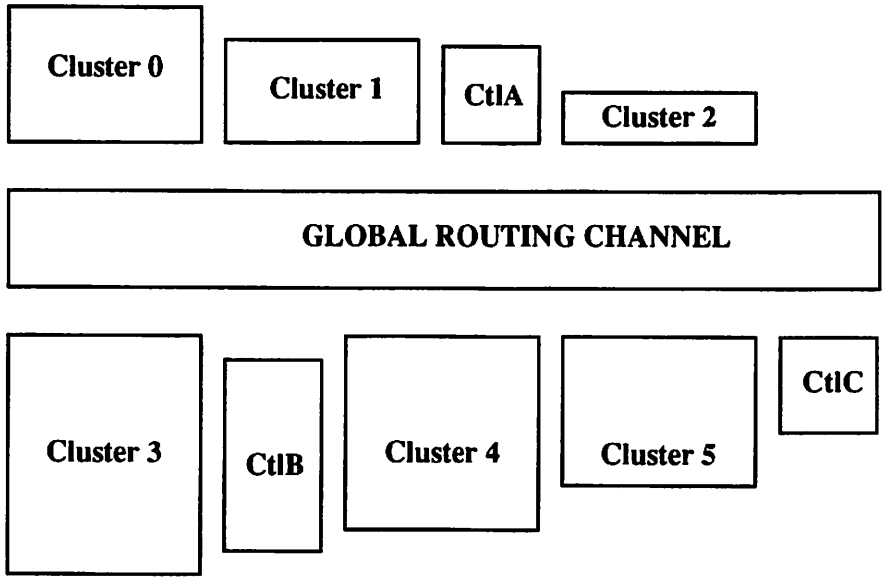

FIGURE 6.3 : Typical set of clusters in a design

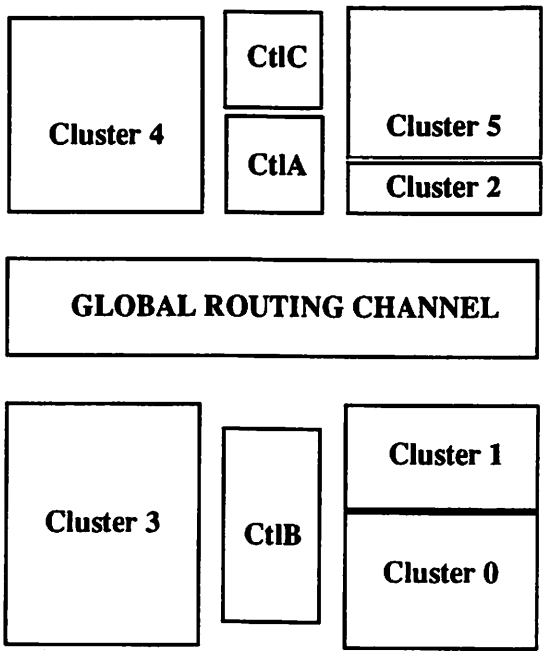

FIGURE 6.4 : Typical set of clusters after merging

6. 2. 2 Automatic Mode

In the automatic mode, the *Mapper*'s merging and floorplanning routines are closely intertwined. The merging occurs first, but it is based on an estimate of the shape and size of the resulting floorplan.

6. 2. 2. 1 Initial solution

An initial solution is derived simply by forming a list of all the units in the design. The units are ordered according to their height, beginning with the tallest. The list is then broken in two at the mid-point, which is determined by the widths of the units. The initial solution typically looks similar to Figure 6.3.

6. 2. 2. 2 The "Merging Factor"

The "Merging Factor" can be specified on the command line when the *Hardware Mapper* is run from a UNIX prompt. It is a real number with a value normally

in the range [1.00 - 1.25]. The *Mapper* uses this factor when deciding which units are candidates for merging. This is further described in the next section.

6. 2. 2. 3 Merging of Clusters

The merging process is simple. It does not always produce optimal results, but it shows that the merging can be done with great improvement in area efficiency. It iterates the following process until the overall chip height-to-width ratio is somewhere in the 50:50 or 40:60 range:

1. Generate an initial solution of an ordered-by-height list of clusters
2. Traverse the list. At each cluster, look forward in the list to find the tallest cluster which, if merged with the current one will not exceed the tallest unit by more than the Merging Factor times the greatest height. If no units are found to satisfy this requirement, continue the traversal of the list. If a unit is found to satisfy the requirement, merge it with the current one and go back to step 1.

6. 2. 2. 4 Automatic Floorplan Generation

Once the merging has been completed the floorplan generation follows naturally. The units are lined up according to their heights, and the line-up is broken at the mid-point as determined by the width of the units. The first half of the line-up is assigned to the lower row of the floorplan (shown in Figure 6.1). The second half of the line-up is assigned to the upper row of the floorplan. Controllers are added to the upper and lower rows and the floorplan is generated in the language *fdl*, the floorplan description language understood by Flint.

6. 2. 3 Manual Mode

The manual mode has three aspects that need to be understood in order to use this *Mapper* feature to the full extent. They include the requirements, the limitations and the specifications.

6. 2. 3. 1 Manual Mode Requirements

The requirements of the manual mode are quite simple. All it requires is that a file name be specified on the command line using the **-f file-name** format. The named file must contain merge and/or floorplan commands in the format described in section 6.2.2.3 below.

6. 2. 3. 2 Manual Mode Limitations

The limitations of the manual mode are numerous. First of all, it does not offer any error checking. If a designer has an error in the command file, the *Mapper* is likely to produce erroneous results. Secondly, the ordering of the merge commands are important. Cluster merging must take place before control block merging. If this is not the case, the results of the merging are unpredictable.

When using the floorplan commands, the designer must take great care to include all units in either the upper list or the lower list. All clusters and control blocks that remain after merging must be listed, and the global controller (named **controller**) must be included as well.

6. 2. 3. 3 Manual Mode Specifications

The command language is very simple. An example will serve to explain the syntax of the language.

Example

The following is the command file for the design shown in Figure 6.5.

```
(     (merge dp5 dp2)
      (merge dp1 dp0)
      (merge Ctl4 Ctl5)
      (merge Ctl1 Ctl3)
      (upper dp4 Ctl4 dp5)
      (lower dp3 Ctl1 controller dp1)
)
```

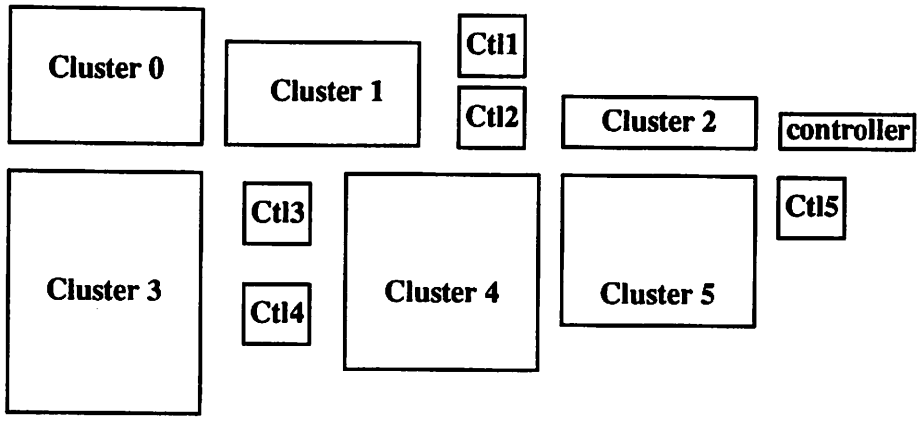

FIGURE 6.5 : Example of blocks for a design

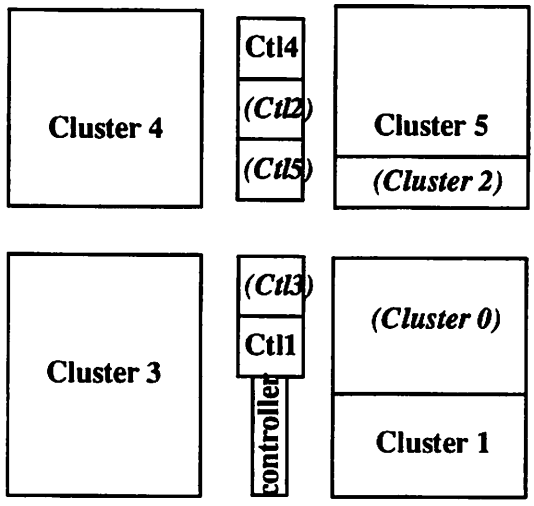

FIGURE 6.6 : The same blocks after applying manual merging.

There are a few points to notice:

1. Notice the three keywords **merge**, **upper** and **lower**. They represent the commands and they must be followed by some arguments.
2. The first command merges dp2 into dp5. After that command executes, dp2 no longer exists in the design. Notice also that the first merge command, even though it only explicitly references the clusters (dp5 and dp2), the controllers that belong to those clusters are

merged together as well. Therefore, after the first merge command has executed, dp2 and Ctl2 are gone, and dp5 and Ctl5 contains the units that used to be in dp2 and Ctl2. Consequently, the names dp2 and Ctl2 can not be used in any subsequent commands.

3. The commands used to merge controllers are printed *after* the merge commands used for the clusters. This is the prescribed order. If this order is not obeyed, the results are unpredictable.
4. All the clusters and control blocks that remain after the merge commands are listed in the upper and lower lists. Notice that the global controller (**controller**) is included.

6.3 The SDL File Hierarchy

The SDL file hierarchy differs from the VHDL file hierarchy in a significant way. The differences stem mostly from implementation considerations or from limitations/features of the Lager tools. This section outlines the elements of the hierarchy, including sections that describe datapaths, arrays, control logic and the global controller. Figure 6.7 shows the hierarchy.

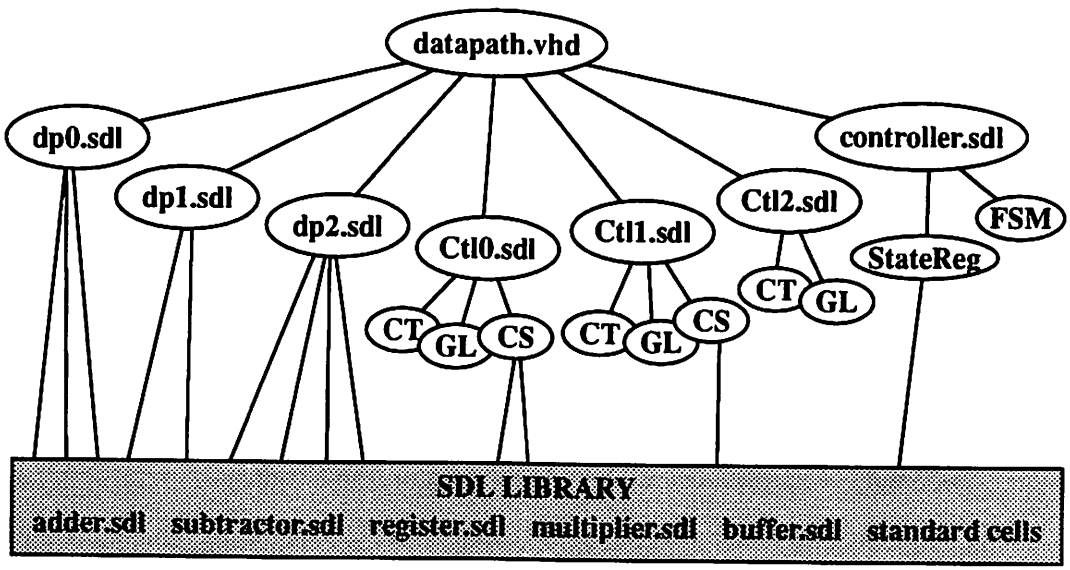

FIGURE 6.7 : The SDL File Hierarchy.

6.3.1 The top level

The highest level in the hierarchy is contained in the file datapath.sdl. This file instantiates all the datapath and array clusters, as well as the global controller and the distributed control logic blocks.

All casting is done at this level. The way casts are handled is described in section 6.4.

6.3.2 Datapaths and Arrays

There is one file for each of the datapaths and arrays in the design. These files are named dp0, dp1, dp2... When cluster merging or control block merging has been performed (as described in section 6.2), the list of names will not be sequential (as in dp0, dp1, dp2), since some of the clusters have "disappeared" by being merged with other units.

In some cases there will be as many clusters as there are execution units in the design. In other cases, e.g. when there are array components in a design, there will be additional clusters added to allow for the separation of arrays and datapaths (see section 6.1.1). Also, if there are various widths in one datapath, extra clusters will be added to allow for the separation of widths (see section 6.1.2). Merging, in contrast, reduces the number of clusters in a design.

The main difference between the SDL files for datapaths and arrays is that datapaths have an *implied* bit width (so the data buses should not be given a bit width), while the arrays must have the bit width stated *explicitly*.

6. 3. 3 Control Logic

The control logic is a two level structure. At its highest level the files are called Ctl0, Ctl1, Ctl2... Each of these files may instantiate one or more sub-controllers. There are three different groups of sub-controllers, each designated for a specific task. The three groups are "control tables" (CT), "glue logic" (GL) and "other logic" (CS, for Control Structure), and they are described below.

When the layout is generated, the control logic is flattened such that it appears to be a single unit. Thus, the separation of the three groups is only seen in their SDL files, not in the final layout.

6. 3. 3. 1 Control Tables

A control table contains information about how control signals should be asserted in all possible control states. The control table generation is described in section 4.6.4. The implementation of control tables exploits the capabilities of the Lager tool called *Bds2stdcell*. It accepts a BDS description [Cas91] (a hardware behavioral language), performs a multiple-level logic optimization with the OCTTOOL misII [Cas91], then maps it to a standard-cell library.

The *Mapper* provides two files for each control table. An SDL file describes the terminal of the block, and a BDS file describes the behavior of the control block. The two files are named e.g. Ctl1CT.sdl and Ctl1CT.bds (where CT is an acronym for Control Table).

6. 3. 3. 2 Glue Logic

The *Mapper* uses glue logic to perform some boolean functions on control signals. One example is when a cell requires both a signal and its inverse (e.g. READ and $\overline{\text{READ}}$) then the control table would typically only represent one of them, and the

glue logic would derive the other (e.g. if the control table contains READ, then the glue logic would have use an inverter to generate $\overline{\text{READ}}$).

As was done for control tables, glue logic is implemented through the tool *Bds2stdcell*. Consequently, the *Mapper* provides two files for each glue logic block, one for the SDL representation and one for the BDS representation.

6. 3. 3. 3 Other Control Logic

Bds2stdcell is limited to pure combinatorial circuits. It can not represent any type of latch or register. The category of "Other Control Logic" was introduced to handle registers and other control logic used in the implementation of control paths. Section 4.5.6 describes the origin of control paths. The *Mapper* generates one SDL file for each "other logic" block, which instantiates the standard cells that were selected during the derivation of the control path.

6. 3. 4 Global Controller

The global controller in the SDL mapped design has two functions. Its primary function is to be the global state counter which synchronizes all the clusters in the design. It also generates the input/output valid flags that help external circuits synchronize with the Hyper generated processor.

The implementation of the global controller has two parts: a control table and a state register. The control table resembles those described in section 6.3.3.1 and it is implemented in a similar way using the SDL/BDS combination. The files that describe the table are FSM.sdl and FSM.bds. The logic that is synthesized from these files generate both the next state and the input/output valid flags.

The state register is made up of standard cell D flip-flops. There are as many flip-flops as ceiling($\log_2$(number-of-states)).

The two components of the global controller are flattened in the layout generation, such that the controller appears to be a single unit.

6.4 Casts

All casting is handled in the top level of the file hierarchy, in the file called datapath.sdl. The concept of cast handling is the same in both SDL and VHDL, but the implementations are quite dissimilar, mainly due to the differing language syntaxes. It is recommended to read section 5.3 before continuing this section.

The cast implementation in SDL is actually more straight forward than in VHDL. The reason for this is that in VHDL new temporary signals have to be made that can accept zero padding or sign/zero extension before being presented to the input terminal of a cluster. In SDL it is allowed to connect the terminal of a cluster with a group of data bits, and then request that some other bits of the terminal be grounded (for zero padding or zero extension) or tied to a sign bit (for sign extension). This is much more convenient than the VHDL method.

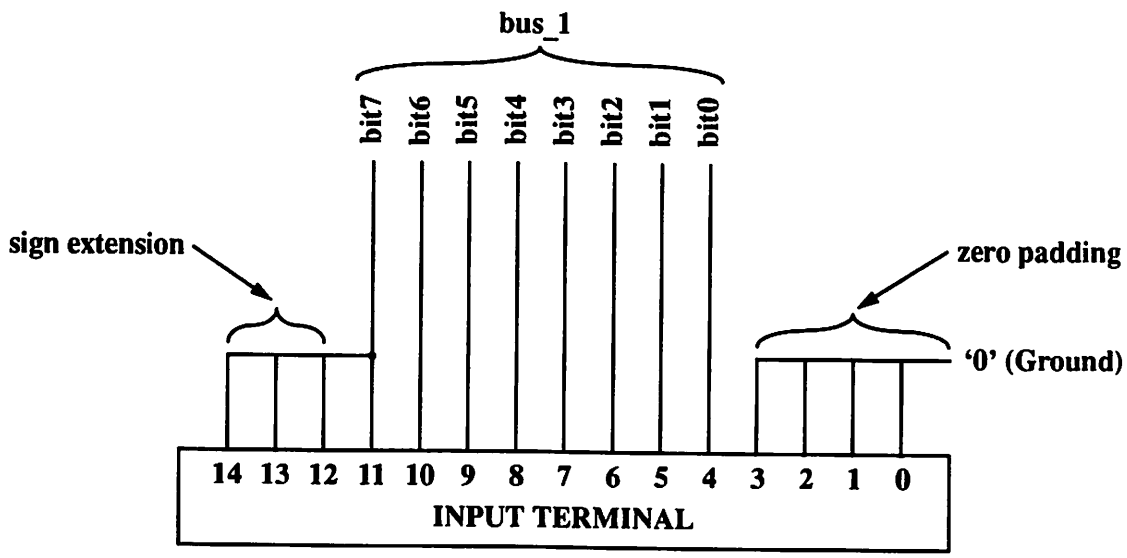

FIGURE 6.8 : Example of a cast in SDL.

7 EXAMPLES

This chapter illustrates some of the functions and features of the *Hardware Mapper*. I will use various implementations for a wavelet filter (a high pass and low pass finite impulse response filter) to highlight various issues. A small implementation is used to show the effects of bus merging. A medium sized implementation is used to show the effects of cluster merging. A version that uses a multiplier instead of constant multiplication expansion is also shown to illustrate the datapath versus array type components.

7.1 The Wavelet Filter Algorithm

This wavelet filter contains a high pass and a low pass FIR filter. The high and low pass sections share their constant coefficients for their 14 taps, but use different addition and subtraction chains. The algorithm shown below, is used for all the examples in this chapter. Figure 7.1 shows a block diagram of the filter, and Figure 7.2 shows its impulse response.

```
#define word fix<17,16>

#define a0 0.015625
#define a1 0.015625
#define a2 -0.046875
#define a3 -0.031250
```

```
#define a4 0.093750
#define a5 0.109375
#define a6 -0.468750
#define a7 0.468750
#define a8 -0.109375
#define a9 -0.093750
#define a10 0.031250
#define a11 0.046875
#define a12 -0.015625
#define a13 -0.015625

func main(In : word) Outhigh, Outlow : word =
begin
    /* Define the high pass filter */
    Acc13 = word(In@13 * a13);
    Acc12 = Acc13 + word(In@12 * a12);
    Acc11 = Acc12 + word(In@11 * a11);
    Acc10 = Acc11 + word(In@10 * a10);
    Acc9 = Acc10 + word(In@9 * a9);
    Acc8 = Acc9 + word(In@8 * a8);
    Acc7 = Acc8 + word(In@7 * a7);
    Acc6 = Acc7 + word(In@6 * a6);
    Acc5 = Acc6 + word(In@5 * a5);
    Acc4 = Acc5 + word(In@4 * a4);
    Acc3 = Acc4 + word(In@3 * a3);
    Acc2 = Acc3 + word(In@2 * a2);
    Acc1 = Acc2 + word(In@1 * a1);
    Outhigh = Acc1 + word(In * a0);

    /* Define the low pass filter */
    Acclow13 = word(In@13 * a13);
    Acclow12 = Acclow13 - word(In@12 * a12);
    Acclow11 = Acclow12 + word(In@11 * a11);
    Acclow10 = Acclow11 - word(In@10 * a10);
    Acclow9 = Acclow10 + word(In@9 * a9);
    Acclow8 = Acclow9 - word(In@8 * a8);
    Acclow7 = Acclow8 + word(In@7 * a7);
    Acclow6 = Acclow7 - word(In@6 * a6);
    Acclow5 = Acclow6 + word(In@5 * a5);
    Acclow4 = Acclow5 - word(In@4 * a4);
    Acclow3 = Acclow4 + word(In@3 * a3);
    Acclow2 = Acclow3 - word(In@2 * a2);
    Acclow1 = Acclow2 + word(In@1 * a1);
    Outlow = Acclow1 - word(In * a0);

end;
```

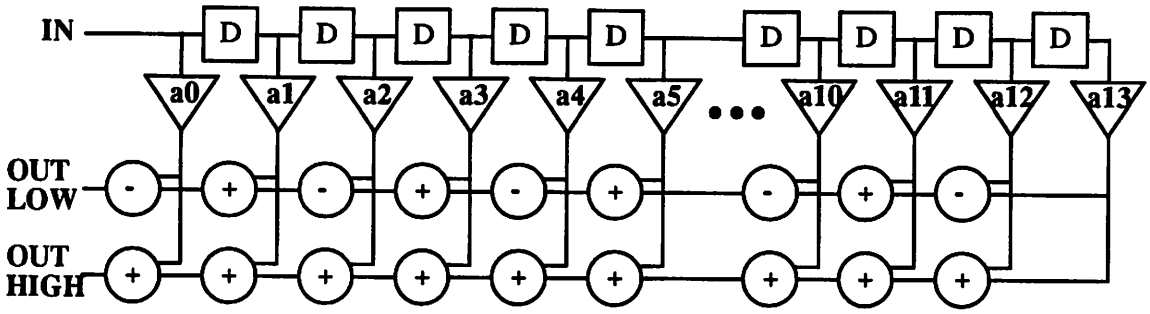

FIGURE 7.1 Block diagram of the wavelet filter.

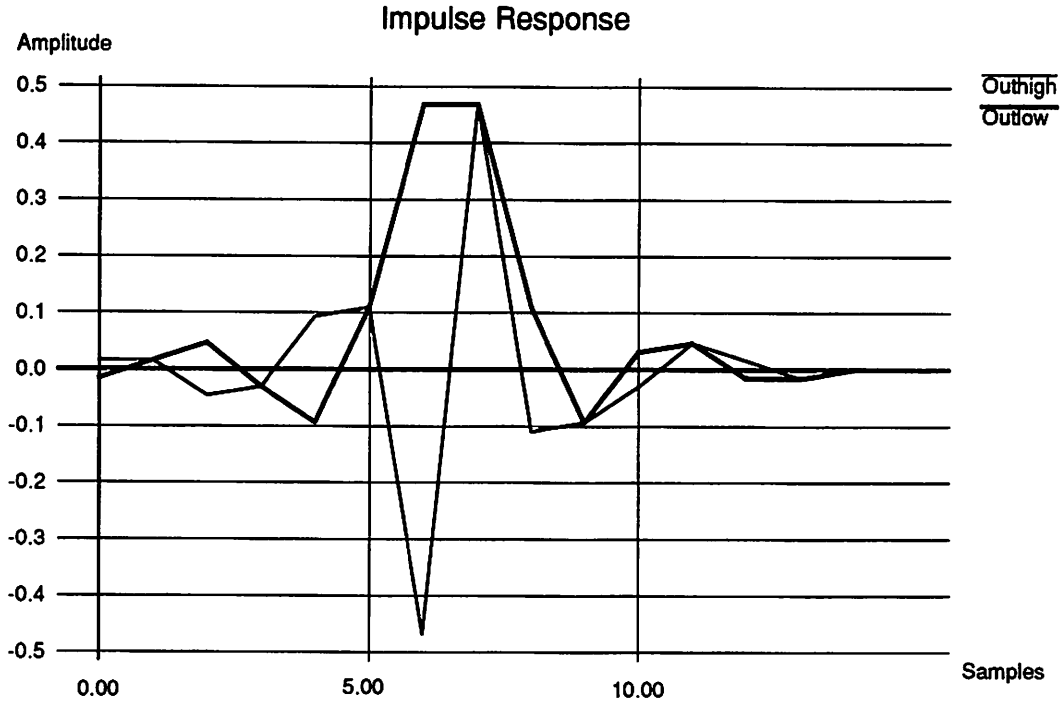

FIGURE 7.2 Impulse response of the wavelet filter.

7.2 Small Wavelet Filter

This implementation has a sample period of 22 clocks/sample. It was generated by Hyper using the following steps:

- parse

- constant multiplication expansion
- module selection
- retiming for speed
- module selection
- estimation
- allocation and scheduling

The estimation of the implementation area calculated by the estimator is

Minimal Active Area : 1.31 mm^2
Total chip area : 6.03mm^2

The scheduler calculated the active area to be 1.88 mm^2.

The schedule of this design is:

EXECUTION UNIT REFERENCE TABLE
1. add#170
2. sub#170
3. ioUnit#170
4. shr#170
5. transfer#170

SCHEDULE

Time	1	2	3	4	5
0	x	x		x	
1		x		x	
2	x			x	x
3	x	x		x	
4	x	x		x	x
5	x	x		x	
6	x	x		x	
7		x		x	x
8	x	x		x	x
9	x	x	x	x	
10		x		x	
11	x	x		x	
12		x		x	

13	x	x	x	
14		x	x	x
15	x	x	x	
16	x		x	x
17		x	x	
18	x	x	x	x
19	x		x	x
20	x	x	x	
21	x		x	x

The first layout (Figure 7.3) shows this design without any bus merging. There are 15 buses, 5 muxes, 18 buffers and 56 registers in addition to the execution units mentioned above. The *Hardware Mapper* predicted the chip area to be 2729 λ x 5245 λ = 5.2 mm^2 (in a 1.2 μm technology). The actual layout dimensions are 4156 λ x 4703 λ = 7.0 mm^2.

The second layout (Figure 7.4) shows this design with global bus merging applied. There are now 5 buses, 4 muxes, 6 buffers (the number of registers is the same). The *Hardware Mapper* predicted the chip area to be 2729 λ x 3658 λ = 3.6 mm^2 (in a 1.2 μm technology). The actual layout dimensions are 3454 λ x 3660 λ = 4.6 mm^2.

The *Mapper*'s area predictions are too small, because the area of control logic and control wiring is not known at the time the prediction is made.

The effect of bus merging is clearly desirable when chip area is a concern. In this example, a savings of 2.4 mm^2 was realized.

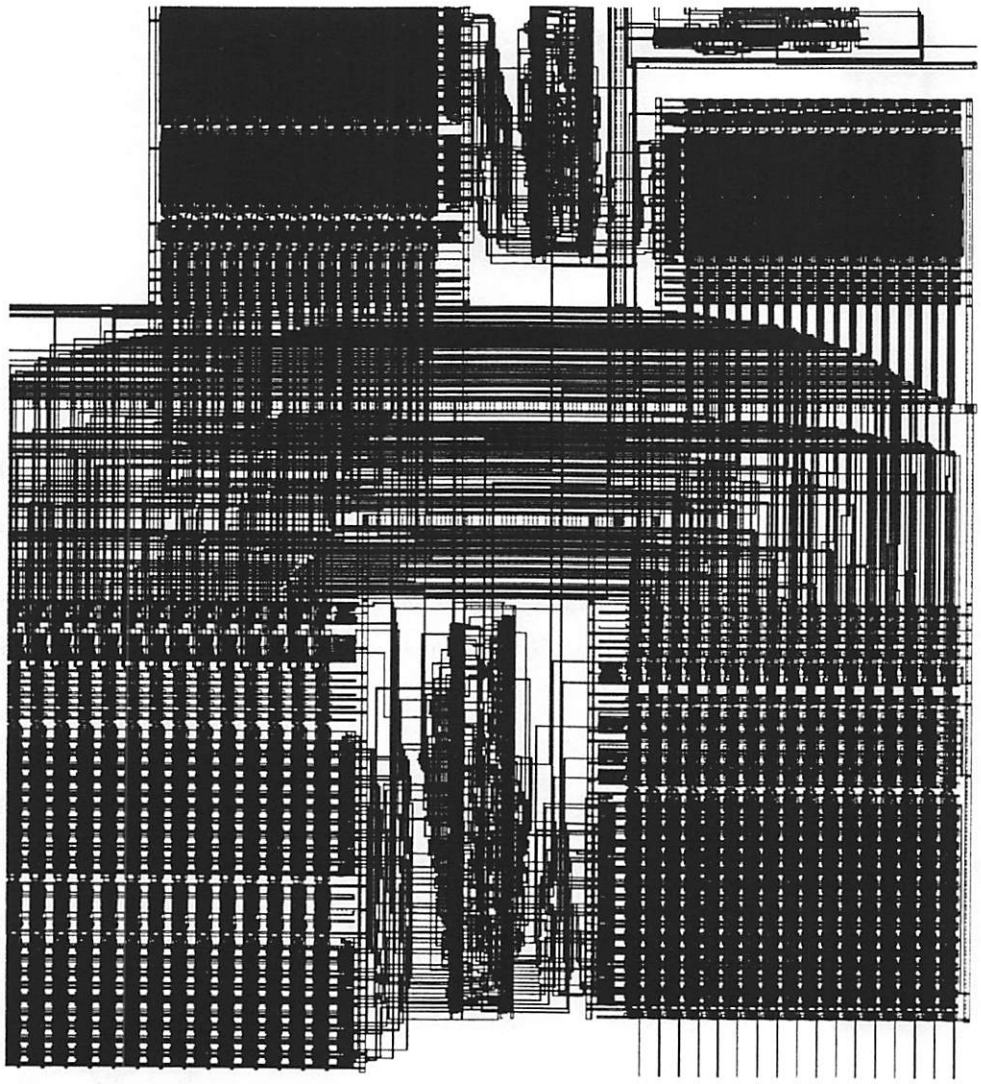

FIGURE 7.3 Small implementation of the wavelet filter without bus merging.

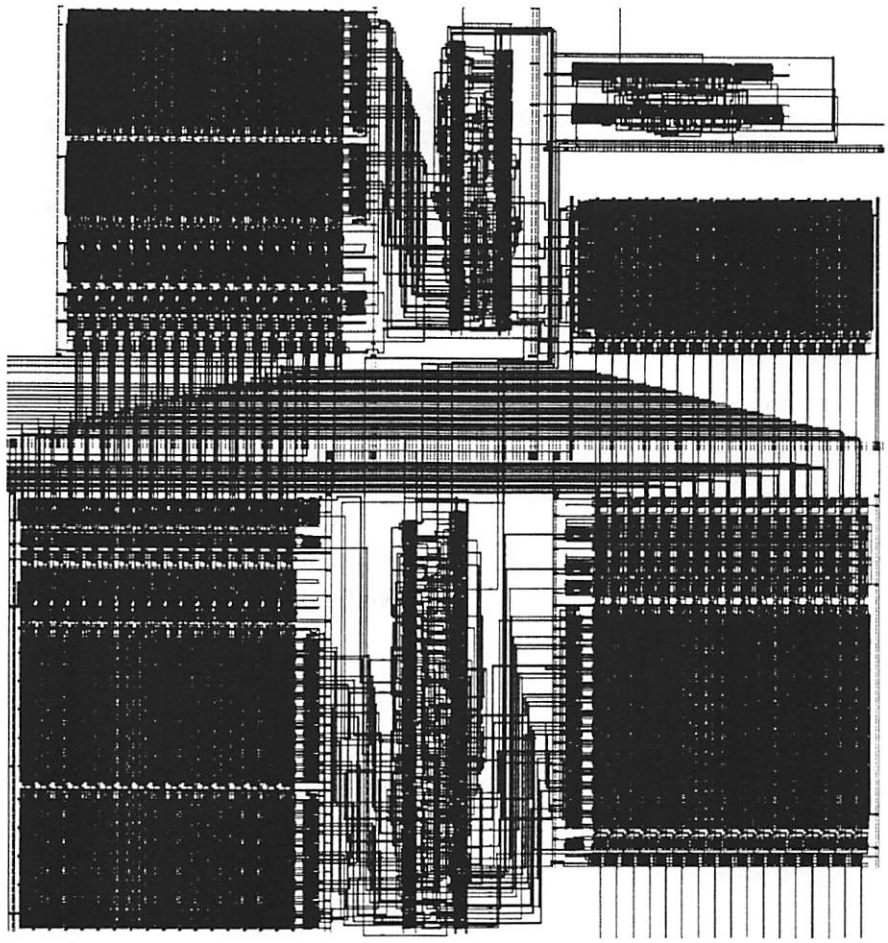

FIGURE 7.4 Small implementation of the wavelet filter with global bus merging.

7.3 Medium Wavelet Filter

This implementation has a sample period of 13 clocks/sample. It was generated by Hyper using the following steps:

- parse
- constant multiplication expansion
- module selection
- retiming for speed

- module selection
- estimation
- allocation and scheduling

The estimation of the implementation area calculated by the estimator is

Minimal Active Area : 1.58 mm^2
Total chip area : 7.72 mm^2

The scheduler calculated the active area to be 2.70 mm^2.

The schedule of this design is:

EXECUTION UNIT REFERENCE TABLE

1. add#170
2. add#171
3. sub#170
4. sub#171
5. ioUnit#170
6. shr#170
7. shr#171
8. transfer#170

SCHEDULE

Time	1	2	3	4	5	6	7	8
0		x	x	x	x	x	x	
1	x			x		x	x	x
2	x		x	x		x	x	x
3	x		x	x		x	x	x
4	x	x		x		x	x	
5		x	x	x		x	x	x
6	x	x				x	x	x
7	x	x	x	x		x	x	
8				x		x		x
9	x	x	x	x		x	x	
10	x	x	x				x	x
11		x	x			x		x
12			x				x	x

The first layout (Figure 7.5) shows this design without any cluster merging. There are 8 clusters plus 8 control blocks. In addition to the execution units mentioned above there are 7 buses, 10 muxes, 18 buffers and 61 registers. The *Hardware Mapper* predicted the chip area to be 6258 λ x 3310 λ = 7.5 mm^2 (in a 1.2 μm technology). The actual layout dimensions are 7082 λ x 3640 λ = 9.3 mm^2.

The second layout (Figure 7.6) shows this design with automatic cluster merging applied. There are now 7 clusters and 4 control blocks. The *Hardware Mapper* predicted the chip area to be 5384 λ x 3447 λ = 6.7 mm^2 (in a 1.2 μm technology). The actual layout dimensions are 6394 λ x 3814 λ = 8.8 mm^2. With some small floorplan adjustments using Flint, the layout area was reduced to 6115 λ x 3745 λ = 8.2 mm^2.

The *Mapper*'s area predictions are too small, mainly because the area of control logic and control wiring is not known at the time the prediction is made.

The effect of merging clusters together is clearly desirable when chip area is a concern. In this example, a savings of 1.1 mm^2 was realized.

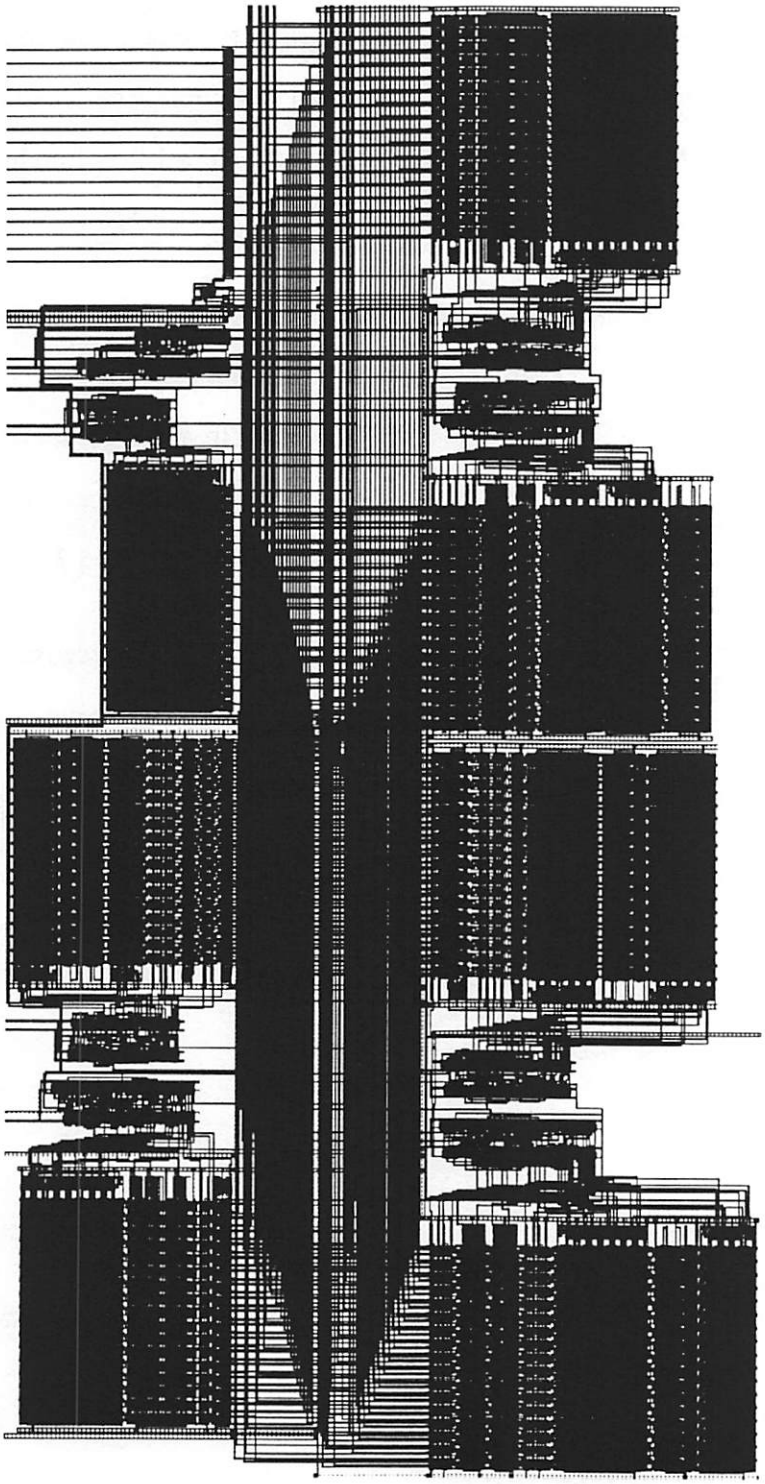

FIGURE 7.5 Medium sized implementation of the wavelet filter without cluster merging.

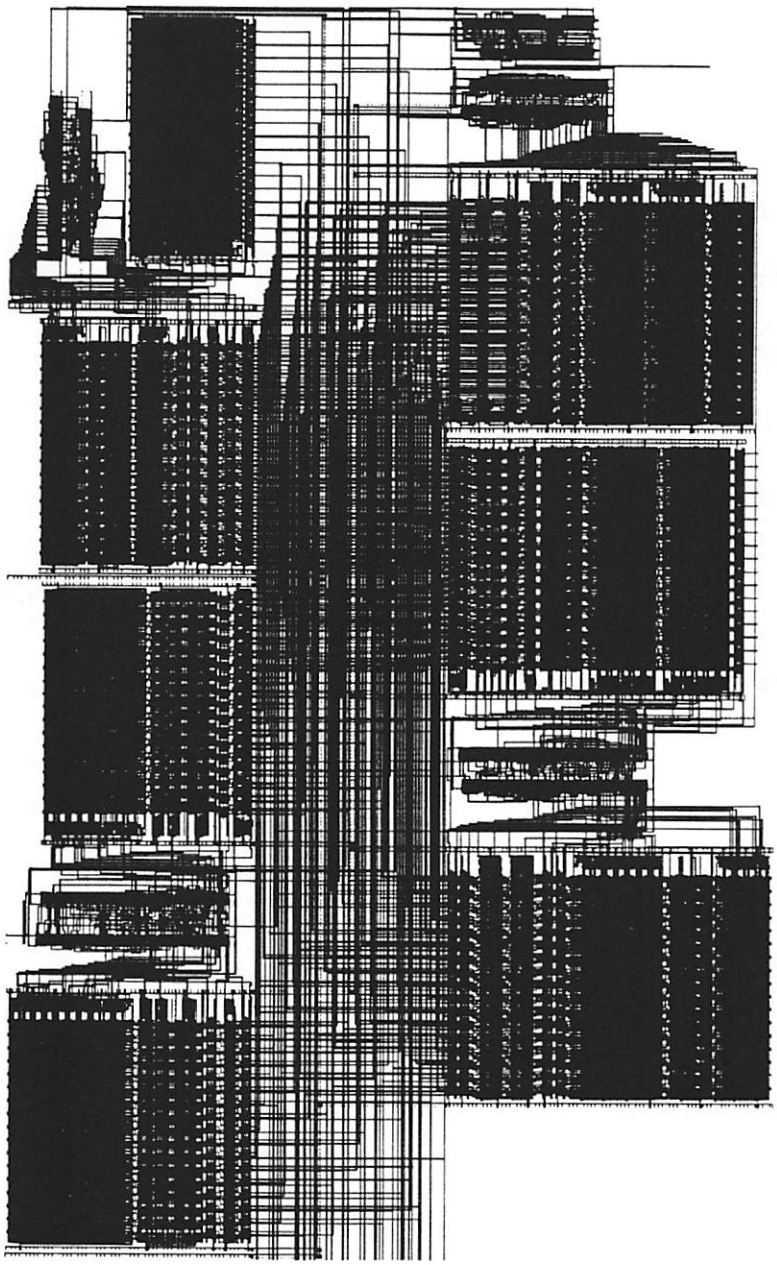

FIGURE 7.6 Medium sized implementation with cluster merging.

7.4 Wavelet Filter With A Multiplier

This implementation has a sample period of 21 clocks/sample. It was generated by Hyper using the following steps:

- parse
- module selection
- retiming for speed
- module selection
- estimation
- allocation and scheduling

The estimation of the implementation area calculated by the estimator is

Minimal Active Area : 4.65 mm^2
Total chip area : 19.10 mm^2

The scheduler calculated the active area to be 5.57 mm^2.

The schedule of this design is:

EXECUTION UNIT REFERENCE TABLE
1. mult#340
2. add#170
3. sub#170
4. ioUnit#170
5. transfer#170

SCHEDULE

Time	1	2	3	4	5
0	x	x	x		
1	x	x			
2	x				
3	x	x			x
4	x	x			

```
5     x  x        x
6     x     x     x
7     x  x        x
8     x  x  x
9     x  x  x
10    x  x        x
11    x  x        x
12    x  x  x     x
13       x  x     x
14       x     x
15    x  x
16       x  x
17       x
18       x
19       x
20       x
---------------------------
```

The layout in Figure 7.7 shows this design. There are 6 clusters plus 5 control blocks. In addition to the execution units mentioned above there are 8 buses, 2 muxes, 6 buffers and 57 registers. The *Hardware Mapper* did not predict the chip area because there is an array component in the design. The layout dimensions are 6603 λ x 3548 λ = 8.4 mm^2.

This example illustrates the way array components are handled. The multiplier in the design is an array component, and as such it is placed in its own cluster. It is evident from Figure 7.7 that the multiplier is much larger than the other clusters. That is the reason the *Hardware Mapper* does not attempt to estimate the area or generate a floorplan.

WIth cluster merging applied, based on a merge file, the layout area can be reduced to 6274 λ x 3465 λ = 7.8 mm^2. This is shown in Figure 7.8.

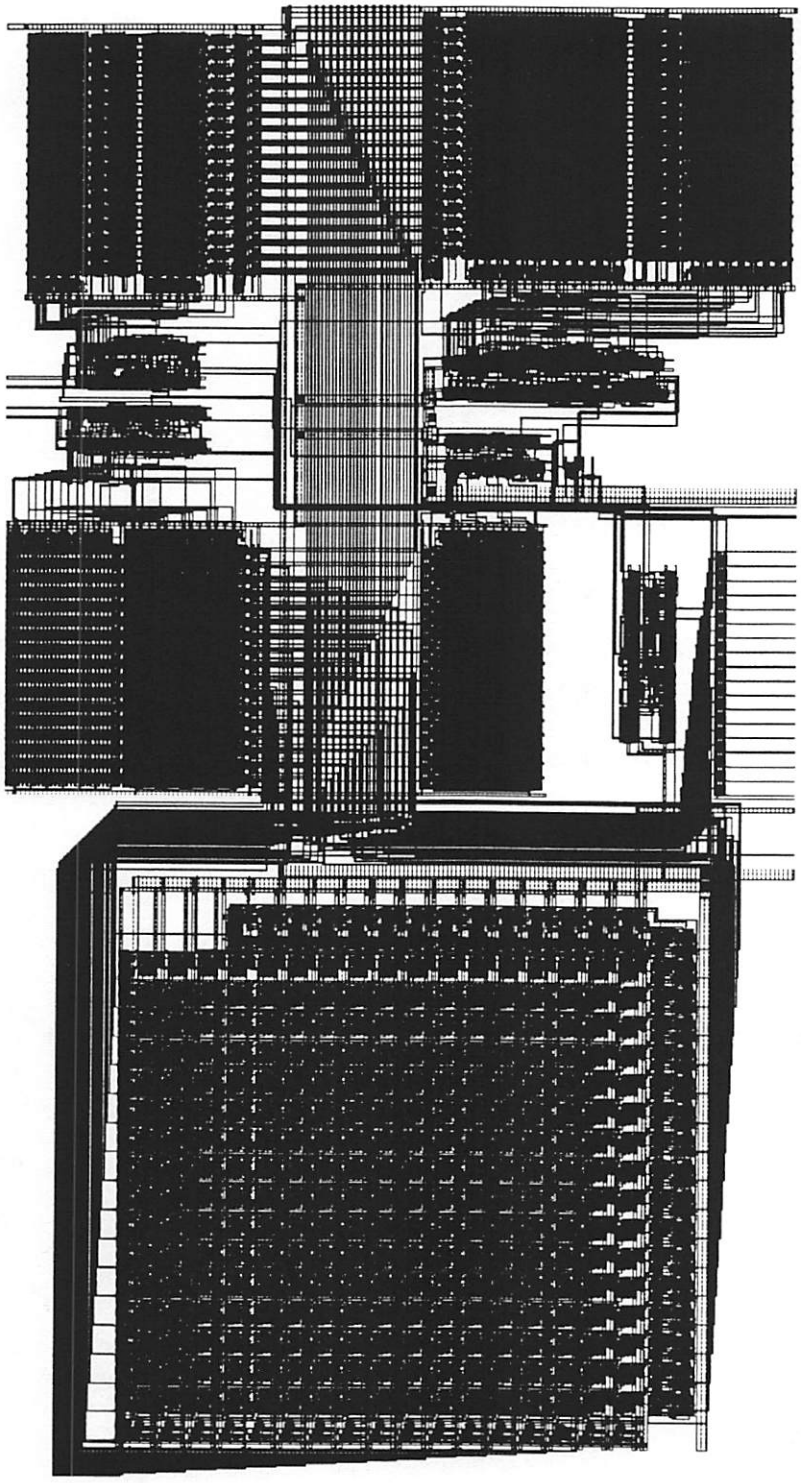

FIGURE 7.7 Wavelet filter implemented with a multiplier (without cluster merging).

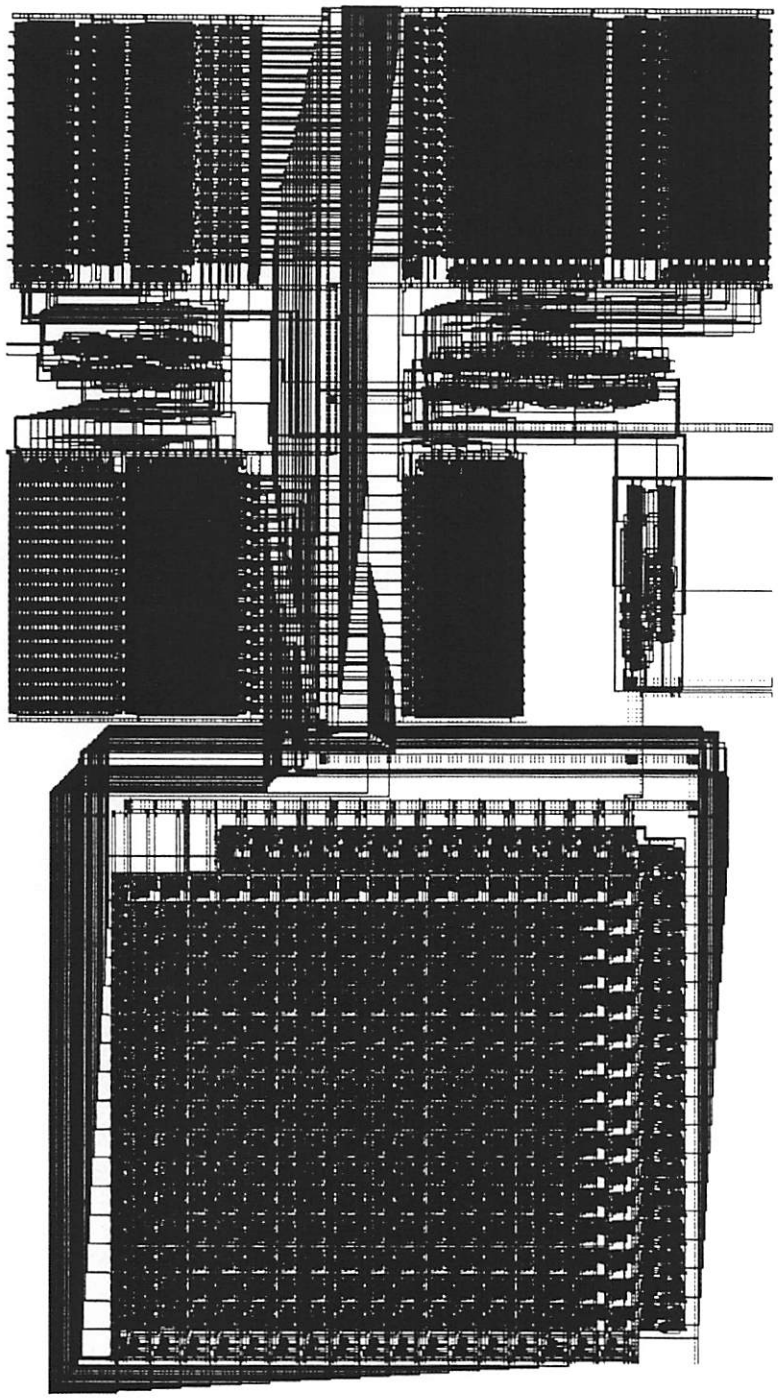

FIGURE 7.8 Wavelet filter implemented with a multiplier (with cluster merging).

8 CONCLUSIONS

A *Hardware Mapper* has been implemented for the Hyper high level synthesis system. The *Mapper* is capable of mapping Hyper designs to two different hardware description languages. The two languages are VHDL, which gives access to commercial layout generators and simulators, and SDL, which gives access to a public domain layout generator.

The *Mapper* has two phases, one that is independent of the targeted hardware description language, and one that is specific to the target. The target independent phase generates a generic design specification based on a Hyper generated flowgraph. The target specific phase translates the generic design to a hardware description.

The *Hardware Mapper* thus completes the automated ASIC design flow, by providing an interface between Hyper and two layout generators.

8.1 Future Work

There are a number of areas in which the *Hardware Mapper* can be extended in the future. As Hyper is extended and gets more and more features the *Mapper* will need to be updated to support them. However, the areas that may need attention in the near term are listed below.

1. The buffer selection routines (section 4.5.1) can be extended to consider the fanout of a bus when determining the sizes of buffers.
2. The automatic mode of the synthesis driven floorplanning (section 6.3) can be extended to consider total chip area including buses, as opposed to just active area.
3. The *Mapper* does not support memories (RAM, ROM etc.) Once Hyper's memory routines are in place, the *Mapper* will need to be updated.
4. Control paths (section 4.6.6) can only have one fanout in the current version of the *Mapper.* Also, only one control path is allowed to control a mux. These limitations can be removed in the future.
5. The *Mapper* has some rudimentary support for "macro nodes". Once Hyper develops full support for this type of node, the *Mapper* will need to be updated.
6. The register selection routines (section 4.2) can be extended to consider power consumption or speed instead of only area. Likewise for the multiplexer and buffer selection routines (section 4.5).
7. The control table optimization (section 4.6.4.2) can be extended to resolve don't cares such as to minimize the boolean equations describing the table. Currently, don't cares can be resolved to the "previous active value".
8. The *Mapper* could be extended to generate chip pads.

Appendix A

USAGE

This appendix will describe how to use the *Hardware Mapper*. The main focus will be on usage of the *Mapper* when invoked from a UNIX command line. Usage from within Hyper's X interface will also be briefly discussed, mainly to outline the subset of features that are available.

1.1 UNIX Command Line

The *Mapper* can be invoked from a UNIX command line. In this mode, the full set of command options are available to the user.

By typing

HwMapper

at a UNIX prompt, the following will appear (if your path includes the Hyper binary directory ($FLOWBIN)):

```
Usage : HwMapper [-abCdDefFiLomMstvw] FlowGraphFile
Arguments :
    -a  : ascii input mode
```

-b : alternative bus terminal placement

-C : number of clocks (default: 2)

-d : debug mode

-D : subdirectory (default: vhdl (or sdl with -s)

-e : edge to trigger state (rising/falling)

-f : floorplan/merge command file

-F : manual Flint placement of final layout

-i : dump internal data structures

-L : cell library

-o : output file (log)

-m : initial bus merging mode

-M : global bus merging

-s : dump sdl file

-t : merging tolerance

-v : verbose mode

-w : warnings

Each of these flags are explained in detail below.

-a : The flowgraph file is in ASCII format. The default is the OCT database format [Har86].

-b : Alternative bus terminal placement. This flag can only be used with the -s flag. The input and output buses of a cluster are placed on opposite sides. The default is to have both input and output buses on the same side.

-C n : Specify the number of clocks. This option is rarely used, since it can be specified for a given library in the technology file. However, if the number of clocks are specified on the command line, then that is the value which is used. Otherwise, if a technology file is available, and if it specifies the number of clocks, then that number is used. Otherwise, there will be two clocks, CK1 and CK2. $n \geq 1$.

-d : Debug mode. This mode produces information helpful for debugging.

-D directory : Specify a sub-directory. All the generated files are stored in directory, except the log file (.hw). The default when using the -s flag is "sdl". The default when not using the -s flag is "vhdl".

-e n : Specify which edge of CK1 should trigger the state transitions. This option is rarely used since it can be specified for a given library in the technology file. If $n = 1$, the state will change on the rising edge of CK1. The default is to change state on the falling edge of CK1.

-f file-name : Specify the file in which to find floorplan and/or merge commands. See section 6.2.2.

-F : Request manual Flint placement of final layout. This disables the automatic floorplanning routines (described in section 6.2.1) while the auto-merging remains enabled.

-i : Dump the complete internal data structure. This option creates a file (with the extension .ids) which contains the complete data structure used for the target dependent phases (SDL or VHDL). This is used only when the mapped result will be showed in an Architectural Drawing (*DrawArch*, not yet released) or in the stochastic power analysis tool *SPA*.

-L library : Specify which cell library to use. See below for examples of usage.

-o logfile : Specify the name of the output log file. The default name is to append *.hw* to FlowGraphFile

-m n : Initial bus merging mode. If n = 1, global buses will be merge at the input of register files. This may eliminate the need for muxes. If n = 2, global buses will be merged at the output of execution units. This may eliminate the need for more than one buffer per execution unit. Only the last -m n on the command line is used. See sections 4.3.1 and 4.3.2.

-M : Perform global bus merging. This can be used with either -m 1 or -m 2. See section 4.3.2.

-s : Generate SDL. The default is to generate VHDL.

-t f : Merging tolerance. When the *Mapper* performs automatic merging of units, it uses a merging tolerance as a guide towards a user's desired layout width-to-length ratios. This option only works when all hardware cells in rb-dp have the properties "HEIGHT" and "WIDTH" in some geometric units (e.g. lambdas or microns are accepted but gates are not). The value for f is typically in the range [1.00 - 1.25].

-v : Verbose mode produces information mainly useful for debugging.

-w : Enable warnings that are non-fatal.

A typical invocation of the *Hardware Mapper*'s VHDL mapping is to use the command

HwMapper -aML dpp fir#3

if a design which has been scheduled (by Hyper) is in the file fir#3.afl. This command reads the design fir#3 from the file fir#3.afl which is assumed to be in ASCII format. It accesses the library specified as *dpp*. It then performs global bus merging, and dumps all the generated VHDL files in a directory called "vhdl". It leaves a log file

in fir#3.hw which contains information about the design, including which buses were left after bus merging, which muxes and buffers were added, which type of hardware units were selected for the register files etc.

A similar command used to generate SDL is

HwMapper -asML dpp fir#3

The difference here is that a directory called "sdl" is created instead of "vhdl", and SDL is generated instead of VHDL.

1.2 Usage within Hyper

The *Hardware Mapper* is accessible from the Hyper window through the button "Map". The Map button pops up a window which shows a few options which correspond directly to some of the command line options mentioned above. There are also a number of options which are passed to the mapper from the general Hyper Options menu. All these options are described below.

1. 2. 1 General HYPER Options

The options that are passed to the *Mapper* from the general Hyper options menu are:

-a : the ascii flag
-L library : the hardware library

1. 2. 2 Specific Mapper Options

The options that are passed to the Mapper from the Hyper Map menu are:

-F : manual Flint placement of the top level hierachy.

-b : alternate bus terminal placement. This flag has no effect when used without the -s flag.

-m n : Initial Bus merging mode: Merge buses at the input of registers (n = 1), or merge buses at the output of execution units (n = 2).

-M : Global bus merging

All the other options are not available from the Hyper window. For the novice *Mapper* user, the options Hyper passes to the *Mapper* are very adequate. For the expert user, the other options are still accessible through command line invocation, and will allow expert judgment to override some of the *Mapper*'s decisions or guide the *Mapper* with regards to certain aspects.

REFERENCES

[Bro92] R. W. Brodersen, editor, "Anatomy of a Silicon Compiler," Kluwer Academic Publishers, Boston, 1992.

[Cas91] A. Casotto, editor, "Octtools 5.1," Electronics Research Laboratory, University of California, Berkeley, 1991.

[Chu89] C. Chu, et all., "Hyper: An Interactive Synthesis Environment for High Performance Real Time Applications", Proc. Int'l Conf. Computer Design, IEEE Computer Society Press, Los Alamitos, Calif., 1989, pp. 432-435.

[Har86] D. Harrison, "Data Management and Graphics Editing in the Berkeley Design Environment," Proc. IEEE Int'l Conf. Computer Aided Design, November 1986.

[Hil85] P. Hilfinger, "SILAGE, A High Level Language and Silicon Compiler for Digital Signal Processing," Proc. IEEE CICC Conf., Portland, May 1985.

[Ker88] B. W. Kernighan, D. M. Ritchie, "The C Programming Language," Prentice Hall, New Jersey, 1988.

[Rab91] J. Rabaey, C. Chu, P. Hoang, M. Potkonjak: "Fast Prototyping of Data Path Intensive Architecture," IEEE Design and Test, vol. 8, no. 2, pp. 40-51, 1991.

[VHD87] "IEEE Standard VHDL Language Reference Manual," Institute of Electrical and Electronics Engineers, Inc., New York, 1987.

[Lager91] "Volume 2: Lager Tool Set," U.C. Berkeley, U.C. Los Angeles, Mississippi State University, Institute for Technology Development, June 1991.