INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600
BEHAVIORAL SYNTHESIS FOR APPLICATION SPECIFIC
PROGRAMMABLE DSP SYSTEMS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By
George S. Powley Jr., M.S.

The Ohio State University
1997

Dissertation Committee:
Dr. Joanne E. DeGroat, Co-Adviser
Dr. Charles A. Klein, Co-Adviser
Dr. Steven B. Biblyk

Approved by

Co-Adviser

Department of Electrical Engineering
ABSTRACT

The behavioral synthesis design methodology has been developed to deal with the growing complexity of designing digital systems. Previous research in the behavioral synthesis of real-time DSP systems has focused on the development of custom, dedicated hardware or embedded software for general purpose DSP microprocessors. This research investigates the application of behavioral synthesis to programmable, application specific DSP cores. The architecture developed for these DSP cores is called Field Programmable DSP Arrays (FPDA). The FPDA architecture can be customized to meet the requirements of the a specific application area, by designing the processor using application specific execution units. The FPDA architecture is programmed using a microcode program. This architecture allows FPDA processors to be included as part of a Application Specific Integrated Circuit (ASIC), or developed as a programmable DSP component. Multi-processor extensions to the basic FPDA architecture are provided, for applications that require additional processing power.

The VDFL behavioral specification language has been developed as the input language for FPDA synthesis. VDFL provides a specification language with language constructs which closely match the characteristics of DSP algorithms. The Wizard behavioral synthesis tool has been developed to synthesize a VDFL specification into a
microcode program for the FPDA architecture. Wizard is designed as an extendible synthesis tool, in order to allow new FPDA processor architectures to be added to the design flow. The FPDA target architectures supported by Wizard include: custom, single-processor, and multi-processor systems. The microcode produced by Wizard can be simulated on a behavioral model of the FPDA processor system. After the implementation of programmable FPDA processors, the microcode could be loaded to an evaluation board for a real-time evaluation of the synthesized system.
ACKNOWLEDGMENTS

I wish to thank my advisor, Joanne DeGroat, for her guidance and support throughout my graduate career at OSU. I am thankful for her confidence and encouragement throughout all of our research projects.

This research was funded by Harris Semiconductor in Melbourne, FL. I wish to thank Harris for supporting this research and Gary Porter for ideas that provided the direction for this research.

I also wish to thank my parents for their encouragement and support throughout my academic lifetime (24 years!).
VITA

October 17, 1968 ....................... Born - Radford, Virginia

1991 ..................................... B.S. Electrical Engineering, The Ohio State University

1992 - 1993 .............................. Research Associate, The Ohio State University

1993 ..................................... M.S. Electrical Engineering, The Ohio State University

1994 - present ......................... Research Associate, The Ohio State University

PUBLICATIONS


FIELDS OF STUDY

Major Field: Electrical Engineering
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>VITA</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>ix</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>x</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Behavioral synthesis</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Motivation for research</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Research objectives and contributions</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Dissertation outline</td>
<td>9</td>
</tr>
<tr>
<td>2. RELATED RESEARCH IN BEHAVIORAL SYNTHESIS</td>
<td>10</td>
</tr>
<tr>
<td>2.1 Behavioral synthesis design flow</td>
<td>11</td>
</tr>
<tr>
<td>2.1.1 Behavioral specification and simulation</td>
<td>11</td>
</tr>
<tr>
<td>2.1.2 Module selection, scheduling, and binding</td>
<td>14</td>
</tr>
<tr>
<td>2.1.3 Code generation</td>
<td>16</td>
</tr>
<tr>
<td>2.1.4 Result validation</td>
<td>16</td>
</tr>
<tr>
<td>2.2 Behavioral synthesis for DSP applications</td>
<td>17</td>
</tr>
<tr>
<td>2.2.1 Synthesis of embedded software</td>
<td>17</td>
</tr>
<tr>
<td>2.2.2 Synthesis of dedicated signal processing architectures</td>
<td>18</td>
</tr>
<tr>
<td>2.3 Conclusions</td>
<td>21</td>
</tr>
</tbody>
</table>
3. OVERVIEW OF THE SYNTHESIS SYSTEM ............................................................... 22
   3.1 FPDA processor design ................................................................................................. 23
   3.2 DSP function design ...................................................................................................... 25
   3.3 DSP system design with FPDA’s .................................................................................. 25
       3.3.1 Behavioral specification and simulation ................................................................ 26
       3.3.2 Behavioral synthesis engine ................................................................................... 27
       3.3.3 Code generation ..................................................................................................... 28
   3.4 Conclusions .................................................................................................................... 29

4. BEHAVIORAL SPECIFICATION AND SIMULATION WITH VDFL ...................... 30
   4.1 Motivation for VDFL specification language .............................................................. 31
   4.2 VDFL language description .......................................................................................... 33
       4.2.1 Lexical elements ..................................................................................................... 34
       4.2.2 Data types ............................................................................................................... 34
       4.2.3 Operators ................................................................................................................ 38
       4.2.4 Expressions ............................................................................................................. 40
       4.2.5 Miscellaneous language constructs ....................................................................... 41
   4.3 Translation of VDFL to VHDL for behavioral simulation ......................................... 41
   4.4 Behavioral simulation .................................................................................................... 44
       4.4.1 Floating-point simulation ....................................................................................... 44
       4.4.2 Fixed-point simulation ........................................................................................... 45
       4.4.3 Simulation result analysis ....................................................................................... 46
   4.5 Conclusions ................................................................................................................... 46

5. FPDA TARGET ARCHITECTURE ................................................................................. 47
   5.1 Generic FPDA architecture .......................................................................................... 48
       5.1.1 Execution units ....................................................................................................... 49
       5.1.2 Register file and connection network ..................................................................... 54
       5.1.3 System controller ................................................................................................... 59
   5.2 Application profiling ..................................................................................................... 61
   5.3 Multi-processor FPDA architecture ............................................................................. 62
   5.4 Conclusions ................................................................................................................... 65

6. BEHAVIORAL SYNTHESIS FOR THE FPDA ARCHITECTURE ........................... 66
   6.1 Design libraries .............................................................................................................. 67
       6.1.1 Target architecture library ..................................................................................... 67
       6.1.2 Function library ..................................................................................................... 69
   6.2 Synthesis algorithms ...................................................................................................... 71
       6.2.1 Module selection .................................................................................................... 73
       6.2.2 Time-constrained scheduling ................................................................................. 74
       6.2.3 Resource-constrained scheduling .......................................................................... 78
       6.2.4 Register binding .................................................................................................... 80
   6.3 Synthesis target architectures ....................................................................................... 84
   6.4 Conclusions ................................................................................................................... 86

vii
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>VDFL lexical elements</td>
<td>36</td>
</tr>
<tr>
<td>4.2</td>
<td>Translation of DFL data type to VDFL data types</td>
<td>36</td>
</tr>
<tr>
<td>4.3</td>
<td>Fixed-point operations in VDFL</td>
<td>38</td>
</tr>
<tr>
<td>6.1</td>
<td>Comparison of scheduling algorithms for a 3rd order IIR filter</td>
<td>78</td>
</tr>
<tr>
<td>7.1</td>
<td>Resource requirements for FIR filter on low-frequency FPDA processor</td>
<td>93</td>
</tr>
<tr>
<td>7.2</td>
<td>Resource requirements for biquad filter on low-frequency FPDA processor</td>
<td>93</td>
</tr>
<tr>
<td>7.3</td>
<td>Resource requirements for FIR filter on high-frequency FPDA processor</td>
<td>96</td>
</tr>
<tr>
<td>7.4</td>
<td>Resource requirements for biquad filter on high-frequency FPDA processor</td>
<td>96</td>
</tr>
<tr>
<td>7.5</td>
<td>Biquad resource usage on the low-frequency FPDA</td>
<td>100</td>
</tr>
<tr>
<td>7.6</td>
<td>Biquad resource usage on the high-frequency FPDA</td>
<td>100</td>
</tr>
<tr>
<td>7.7</td>
<td>Cycles required for the parallel biquad system using one FDPA</td>
<td>100</td>
</tr>
<tr>
<td>7.8</td>
<td>Cycles required for the parallel biquad system using multiple FPDA's</td>
<td>103</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Design descriptions and level of abstraction</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Behavioral synthesis design flow</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>Wizard behavioral synthesis design flow</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>VDFL signal representations</td>
<td>37</td>
</tr>
<tr>
<td>4.2</td>
<td>Sample delay in VDFL and VHDL</td>
<td>39</td>
</tr>
<tr>
<td>4.3</td>
<td>VDFL translation</td>
<td>43</td>
</tr>
<tr>
<td>5.1</td>
<td>The generic FPDA architecture</td>
<td>48</td>
</tr>
<tr>
<td>5.2</td>
<td>Processing element architecture</td>
<td>50</td>
</tr>
<tr>
<td>5.3</td>
<td>Execution unit architecture</td>
<td>51</td>
</tr>
<tr>
<td>5.4</td>
<td>An EU record from the execution unit library</td>
<td>53</td>
</tr>
<tr>
<td>5.5</td>
<td>Register file block diagram</td>
<td>56</td>
</tr>
<tr>
<td>5.6</td>
<td>Internal connection network for a single-processor FPDA</td>
<td>58</td>
</tr>
<tr>
<td>5.7</td>
<td>Microcode program controller</td>
<td>60</td>
</tr>
<tr>
<td>5.8</td>
<td>FPDA multi-processor topology</td>
<td>63</td>
</tr>
<tr>
<td>5.9</td>
<td>Internal connection network for a multi-processor FPDA</td>
<td>64</td>
</tr>
<tr>
<td>6.1</td>
<td>A target record from the target architecture library</td>
<td>68</td>
</tr>
</tbody>
</table>
6.2 A function record from the function library .........................................................69
6.3 Time constrained scheduling algorithm used by Wizard .....................................75
6.4 Resource constrained scheduling algorithm used by Wizard ..............................79
6.5 Modified left-edge algorithm for register binding ...............................................82
6.6 Register binding using the modified left-edge algorithm ....................................84
7.1 VDFL description of an 8th order symmetric FIR filter ......................................89
7.2 VDFL description of a biquad filter .....................................................................90
7.3 Maximum frequency response of FIR filter on low-frequency FPDA ..............94
7.4 Maximum frequency response of biquad filter on low-frequency FPDA ............94
7.5 Maximum frequency response of FIR filter on high-frequency FPDA .............97
7.6 Maximum frequency response of biquad filter on high-frequency FPDA ..........97
7.7 Wizard schedule display for biquad filter ............................................................99
7.8 System of parallel biquad filters ......................................................................102
CHAPTER 1

INTRODUCTION

Behavioral synthesis is a design methodology that transforms a behavioral description of a digital system into a structural description. The structural description may contain hardware, software, or a combination of both. Research in the area of behavioral synthesis for DSP systems has addressed many target architectures, from custom, dedicated signal processors to general purpose DSP microprocessors. This research strives to expand the application of behavioral synthesis to include a new architecture, application specific Field Programmable DSP Array (FPDA) processors. The FPDA architecture will include both single processor and multi-processor systems.

The first section of this chapter introduces the behavioral synthesis design methodology. The second section provides motivations for this research. The third section describes the objectives of the research. The last section provides an outline for the dissertation.
1.1 Behavioral synthesis

As VLSI technology continues to increase the number of transistors that can be fabricated on a single chip, the complexity of digital systems also increases. Behavioral synthesis addresses the complexity of designing digital systems, by automating system design starting from a higher level of abstraction, the behavioral specification [1]. A behavioral specification describes the functionality of a system, but does not contain any implementation decisions. The ultimate goal of the behavioral synthesis design methodology is to automate the design process from system specification all the way down to implementation [2-4], which consists of hardware, software, or a mixture of both.

Synthesis is defined as a translation of a behavioral description into a structural description. The diagram in Figure 1.1 shows the three domains of design description: behavioral, structural, and physical. As the description moves away from the center of the graph, the level of abstraction increases. Behavioral synthesis, also called high-level synthesis or architectural synthesis, starts with a behavioral description at a high level of abstraction. This behavioral description is translated into a structural description at a high level of abstraction. At this point, other tools are required to carry the design to implementation.
Behavioral synthesis tools have usually focused one of three application areas: general-purpose applications, control-dominated applications, and datapath intensive applications. By focusing on one of these areas, the target architecture of the synthesis
tools can be optimized to match the application. Digital signal processing (DSP) systems fall into the datapath intensive category.

The need for high performance dedicated signal processors, operating at high sample rates has attracted the development of many behavioral synthesis tools for DSP applications. The target architecture supported by most of these tools is a custom, dedicated signal processor, which is fabricated as an Application Specific Integrated Circuit (ASIC). Another target supported by DSP synthesis tools is general purpose DSP microprocessors. The output of behavioral synthesis for a DSP microprocessor target is a high-level language program or a machine code program for the desired behavior.

The ability to design application specific signal processors is a major advantage of behavioral synthesis, but an ASIC implementation is not always the best choice for implementation. The are several potential disadvantages to using an ASIC. For low volume manufacturing, the cost of developing an ASIC is high. The modification of a production system which uses an ASIC is also costly. Using an ASIC implementation complicates building prototype systems, which are useful when the complexity of the DSP system prevents efficient simulation. General purpose DSP microprocessors address some of these concerns, but they are not application specific by design. A FPDA target architecture would cover the design areas left open by ASIC and general purpose DSP microprocessor target architectures.
1.2 Motivation for research

The use of field programmable devices offer several advantages in the design of digital systems. First, they provide the ability to easily modify a production system. Second, they provide faster turn-around times compared to spinning an ASIC. Third, they can be used in an evaluation system to provide a real-time system prototype. Finally, they are more cost effective for manufacturing low volume devices. The motivation of this research is to expand behavioral synthesis to target an application specific FPDA architecture.

Since digital signal processing can be applied to many different application areas, the behavioral synthesis system should support the design of application specific FPDA processors. Application specific processors allow the DSP system to be tailored to meet the requirements of the application area. For example, a low-power FPDA processor could be developed for use in portable devices, or a multi-processor FPDA system could be developed for high-throughput applications. The behavioral synthesis system should be extendible, so new processors can be developed as new application areas are created.

The FPDA architecture is not limited to field programmable devices. An FPDA processor could be implemented as a DSP core. In this format, the processor would be considered a mask programmable device, or MPDA. The MPDA would allow application specific circuitry, such as analog signal processing, to be added to the component. Throughout this dissertation, FPDA will be used to refer to both field programmable and mask programmable devices.
For input to the behavioral synthesis system, a machine readable, simulatable language provides additional advantages to the behavioral synthesis design methodology [5]. First, the simulatable specification can be used to verify that the functionality of the design meets the designer’s expectations. Second, the simulatable specification will be more precise, compared to an English specification. Ambiguities which might exist in an English specification would be eliminated by the simulation semantics of a simulatable specification language. Lastly, the simulatable specification can be used to validate synthesis results, by comparing simulation results at different levels of abstraction.

There is a direct correlation between the behavioral specification language used to represent a design, the actual design, and the technology used to implement the design. Therefore, the choice of a behavioral specification language can greatly affect the behavioral synthesis results. To provide optimum synthesis results, the behavioral specification language should provide a direct mapping of design characteristics in the target application to language constructs in the specification language.

1.3 Research objectives and contributions

The overall goal of this research is to expand behavioral synthesis to support application specific FPDA processor architectures. As the DSP systems continue to be applied to many different application areas, the need for target architectures other than ASIC’s and general purpose DSP microprocessors increases. The specific goal of this research is to design a behavioral synthesis system which enables the design of application
specific FPDA processors and generates the microcode required to program these processors. This research involves the development of a flexible behavioral synthesis system, which can be extended to include application specific execution units, and can be used to target different FPDA processors. This dissertation describes the Wizard behavioral synthesis system, which has been designed to meet these research goals.

Wizard addresses the issue of supporting application specific execution units through the use of the execution unit library. Wizard provides the designer with the capability of adding predesigned, possibly proprietary, application specific execution units to the behavioral synthesis design flow. There are several advantages to incorporating this feature into the behavioral synthesis design methodology. First, it allows the synthesis tool to draw from the creativity and experience of the designer. By reusing execution units, the synthesis design flow benefits from the design and testing of previous product designs. Also, the system designer will probably have more design insight in the application area than the designer of the synthesis tool's execution unit library. Second, execution units can be tailored to the application area of the system being designed, such as high-performance vs. low-power. Finally, the synthesis tool can grow as technology continues to advance and new application areas are created.

Wizard addresses the issue of applying behavioral synthesis to FPDA processors through the use of a target architecture library. The target architecture library is a user extendible library which guides the behavioral synthesis process. An entry in the target architecture library describes the operation mapping and resource constraints for a FPDA
processor. By using an extendible target architecture library, Wizard can support new
FPDA processors as they are developed.

The contributions of this research to the DSP behavioral synthesis design
methodology include:

1. The application specific Field Programmable DSP Array (FPDA) processor
target architecture. This target architecture provides a flexible processor
design, which can be extended to include application specific execution units.
The FPDA architecture also include extensions for multi-processor designs,
which can be used to increase the computation power of a design.

2. Behavioral specification of DSP systems using the VHDL Data Flow Language
(VDFL). The VDFL language constructs have been designed to match the
design characteristics of DSP algorithms. This language provides an optimized
input language to the behavioral synthesis process. VDFL is also translated to
VHDL in order to perform behavioral simulations.

3. The Wizard behavioral synthesis tool, which is used to develop FPDA
processors, as well as to generate the microcode for FPDA systems. Wizard
provides four synthesis engines to support these design activities. One engine is
used to profile DSP algorithms during the development of a new FPDA
processor. Another engine is used to generate FPDA microcode for a single
processor system. The last two engines are used to generate FPDA microcode
for multi-processor systems.
1.4 Dissertation outline

The organization of the dissertation is described below:

- Chapter 2 describes previous research in the area of behavioral synthesis. The behavioral synthesis design flow is presented, as well as previous research in the area of real-time DSP synthesis.
- Chapter 3 provides an overview of the behavioral synthesis design flow used in the Wizard synthesis tool.
- Chapter 4 introduces the VDFL behavioral specification language, which is used as the input to behavioral synthesis.
- Chapter 5 describes the generic single-processor FPDA target architecture. Extensions to the architecture to support multi-processor designs is also presented.
- Chapter 6 describes the Wizard synthesis engine. The design libraries and algorithms used during the behavioral synthesis process are detailed.
- Chapter 7 provides a design examples using the Wizard behavioral synthesis tool.
- Chapter 8 provides some conclusions and future directions for the FPDA design methodology.
CHAPTER 2

RELATED RESEARCH IN BEHAVIORAL SYNTHESIS

Unlike logic synthesis, which is based on the manipulation of Boolean equations, behavioral synthesis does not have a formal model. Previous research in behavioral synthesis has provided a common model of the behavioral synthesis process, which can be applied to different application areas. Most of the research in the behavioral synthesis of real-time DSP systems has focused on the development of custom, dedicated signal processing architectures. In this chapter, the common model of the behavioral synthesis design process is described. This common design process has been incorporated into this research. Previous research in the behavioral synthesis of DSP systems is also described, with emphasis on the ideas expanded by this research.

The first section of this chapter describes the common behavioral synthesis design flow. The second section discusses previous research in the behavioral synthesis of DSP systems.
2.1 Behavioral synthesis design flow

Previous research in behavioral synthesis has provided a common model of the synthesis design flow. The steps in this common model include: behavioral specification, behavioral simulation, module selection, scheduling, binding, code generation, and result validation. The design flow using these steps is shown in Figure 2.1. In this section, the common model of a behavioral synthesis design flow is described.

2.1.1 Behavioral specification and simulation

The first step of the behavioral synthesis process is to capture the design specification. There are a wide range of methods used to perform this function. Previous behavioral synthesis tools have used HDL's, applicative languages, and signal flow graphs. Examples of HDL behavioral descriptions include VHDL [6] and Verilog [7]. The advantage of using an industry standard HDL is the ability to develop and test the design using high quality simulation tools available from a number of different tool vendors. A disadvantage of using an HDL is that a programming language might not be the most natural way to describe the behavior of the system, such as the description of digital filters.
Figure 2.1: Behavioral synthesis design flow
Applicative languages, such as Silage [8], allow the designer to specify the behavior of the system using mathematical notation. Each line of a Silage program is treated as an independent function, as opposed to the execution of a sequential program, where the overall behavior of the system is based on the sequence of the operations. Using an applicative language reduces some of the programming overhead incurred when using an HDL, but the development tools available are usually limited to those provided by the behavioral synthesis tool.

Schematic capture is used to generate a signal flow graph representation of a behavioral specification. For some applications, a signal flow graph is a very natural input language for the behavioral description, for example digital filters. A disadvantage of schematic capture is the non-portable nature of schematic entry software. Signal flow graphs are also used as the intermediate format for almost all behavioral synthesis tools [3]. Tools that use an HDL or Silage format, convert these descriptions into signal flow graphs before performing the synthesis operations.

Behavioral simulation is performed on the specification in order to verify that the description satisfies the design requirements. Previous behavioral synthesis tools have used HDL simulators [9], custom simulators [10], or compiled programs to perform the behavioral simulations. The most desirable solution is to use an industry standard HDL, which will allow the simulation to be performed using powerful simulation tools on different platforms and with varying tool vendors.

The results of the behavioral simulation should be viewed in a format that matches the system application. For example, a frequency response plot would be a
natural format to view the simulation results of a digital filter. Some behavioral synthesis tools provide custom simulation, as well as custom result viewers. A desirable solution to viewing simulation results is to use external data analysis tools. This allows the designer to utilize tools they are accustomed to using and present the simulation results in a meaningful format. Using external data analysis tools also allows the behavioral synthesis methodology to grow as it is applied to new application areas.

2.1.2 Module selection, scheduling, and binding

The tasks of module selection, operation scheduling, and resource binding are the heart of the behavioral synthesis process. Many different algorithms have been used to accomplish each of these tasks. Since most of the problems solved by behavioral synthesis are NP-complete [11], these algorithms make a tradeoff between computational complexity and the optimization of the results.

The module selection step matches operations in the behavioral description to execution units in the execution unit library. When there is more than one type of execution unit to perform the same operation, such as different adder architectures, module selection can be used to perform design space exploration. Design space exploration weighs the tradeoffs of area vs. time when designing a custom architecture. For target architectures that only have one execution unit to perform each operation, module selection performs a direct mapping. Module selection also assigns the number of cycles required to execute each operation.
The next step in the synthesis process is scheduling, one of the most widely researched areas of behavioral synthesis. During scheduling, each operation in the signal flow graph is assigned to a control step, or clock cycle. The problem of assigning all operations to a minimum number of resources is an NP-complete problem [12]. Previous approaches to the scheduling problem include [13]: heuristics, list-based scheduling [14], force-directed scheduling [15], integer programming, simulated annealing [16], and continuous relaxation techniques [17]. All of these approaches have found some level of success in previous research systems. The real test of a synthesis system is the amount of time it takes to produce near optimal results on a real world problem.

The binding step maps each operation to a specific hardware instance. In some synthesis tools, binding and scheduling are combined into one step, or there order is interchanged. Resource binding is used to perform tradeoffs between execution units, registers, and interconnect buses. Register assignments are also made during the binding process. The number of registers used is minimized using lifetime analysis techniques. At this point of the synthesis process, the system contains all information needed to pass the design to implementation tools. The next step in the behavioral synthesis design flow is code generation.
2.1.3 Code generation

Code generation involves the creation of a design description which can be passed to implementation tools. The implementation tools supported by behavioral synthesis include: logic synthesis tools, high-level compilers, and microprocessor assemblers. Logic synthesis tools are used to create custom processors, which could be implemented using an ASIC or FPGA design. High-level compilers are used to compile high-level languages, such as C, for a general purpose processor. Some behavioral synthesis tools improve the quality of synthesizing to a general purpose processor, by directly generating assembly language.

2.1.4 Result validation

Result validation is the process of using simulation to validate the results of the synthesis process. The behavioral synthesis process is validated by comparing the results of the behavioral simulation to a simulation of the synthesized system.

Since the behavioral simulation does not contain as much information as the simulation of the synthesized system, a waveform comparison at each system clock will not verify the quality of the design [2]. For DSP applications, the solution would be to compare the output of the RTL simulation to the behavioral simulation at each sample clock, which will verify the input-output characteristics of the system.
2.2 Behavioral synthesis for DSP applications

Research in the behavioral synthesis has produced a large number of synthesis systems [9, 10, 14, 15, 17-26]. The need for high performance, dedicated signal processors, operating at high sample rates has attracted the application of behavioral synthesis to real-time digital signal processing (DSP) applications. Most synthesis tools produce excellent results in the form of custom, dedicated signal processing architectures. Other researchers have focused on the application behavioral synthesis to generate embedded software for general purpose DSP microprocessors.

Behavioral specification methods used by other high-level synthesis systems include VHDL [6], Verilog [7], Silage [8], and graph models. Since the high-level synthesis process starts from a behavioral specification, the choice of a specification language has a direct impact on the quality of the synthesis results. To provide optimum results, the chosen specification language should provide a direct mapping of design characteristics in the target application to language constructs in the specification language [5].

2.2.1 Synthesis of embedded software

Traditionally, general purpose DSP systems have been programmed by hand, using assembly language. Some DSP behavioral synthesis have taken the approach of automatically generating code for general purpose DSP microprocessors. These tools
assert that compilers for imperative language, such as C or C++, produce inefficient code that effects the feasibility of real-time implementation.

Ptolemy [21] is the most successful example of a tool which generates software for general purpose DSP microprocessors used in embedded systems. Ptolemy uses a block diagram language to provide a formal model of the DSP system, which is then compiled into assembly language for the target processor. Ptolemy uses a library of predefined and prescheduled actor code blocks which are used in the SDF specification. This idea is incorporated into the FPDA behavioral synthesis system in the form of DSP functions. DSP functions are prescheduled DSP algorithms which are stored in a library. These functions are used to reduce the amount of interprocessor communication required by a multi-processor FPDA system.

2.2.2 Synthesis of dedicated signal processing architectures

Most of the research in behavioral synthesis has concentrated on the area of high-performance, dedicated DSP architectures. Behavioral synthesis of DSP architectures focuses on the optimization of expensive execution units, such as adders and multipliers, therefore, early researchers saw the importance of this area. Many tools have been developed to address this area, including Cathedral [22], MARS [23], and HYPER [10].

The Cathedral system contains two target architectures: microcoded processors (MP) and hardwired lowly multiplexed data paths (HLMD). The MP architecture
contains a network of execution units, a microcode controller, and a centralized RAM. The MP architecture style is well suited for signal processing algorithms with sample frequencies in the range of 1kHz to 1 MHz. The HLMD architecture contains a network of execution units with a finite-state machine controller, and distributed memories. This architecture style is well suited for signal processing algorithms with sample frequencies above 1 MHz. The HLMD architecture closely resembles the architecture that is used for FPDA processors. The FPDA employs a programmable microcode controller, and provide architecture extensions to support multi-processor systems.

The MARS system utilizes a unified scheduling and binding algorithm, based on a list-based algorithm that schedules the loops in the signal flow graph. This scheduling algorithm allows a flow graph to be scheduled using a iteration period less than the critical path time, by automatically retiming the graph and introducing pipelining stages. Since, the target architecture of MARS is a fully custom network of adders and multipliers, this scheduling algorithm can not be directly applied to the FPDA processor. The use of a list-based scheduling and binding algorithm, which provides very good results in a short amount of computation time, prompted the exploration of a list-based algorithm for the FPDA architecture.

Multi-processor systems provide the advantage improved system throughput over single-processor implementations [24, 25]. The technique of using multi-processing systems to improve computation power has also been an active research area. Most research efforts in this area have been focused on using general purpose
DSP microprocessors, which can limit the throughput of the system due to using asynchronous interrupts for communication. A more efficient approach would be to develop a network of custom data paths, which maximizes throughput using synchronous interprocessor communication. This is the approach that is taken in the FPDA multi-processor architecture.

HYPER [10] is a synthesis environment that targets applications in telecommunications, speech, video, and image processing. The target architecture of HYPER is application specific data paths, connected by a network of high bandwidth buses. HYPER uses Silage for its input language, which is converted to an internal Control Data Flow Graph (CDFG). Behavioral simulations are performed by converting the CDFG into executable C code. Since Silage was designed to describe DSP systems, it is a natural language for the behavioral specification of DSP algorithms. This research develops a behavioral specification language based on Silage and VHDL. This language takes advantage of the strengths of Silage as DSP description language and the strength of VHDL as a simulation language.

HYPER’s approach to scheduling and binding is to perform assignment, or binding, before scheduling. Binding is accomplished using an iterative probabilistic approach, using a quality measure to predict the chances of finding a successful schedule. After the assignment is accepted, scheduling is performed, using resource utilization as the priority function. For the FPDA architecture, scheduling and binding are performed simultaneously using a list-based algorithm.
The system controller generated by HYPER is a synthesizable state machine using a very large instruction word format. A finite state machine description is generated, and synthesized using the MIS-II logic synthesis environment. The output of the HYPER synthesis tool is an SDL description of the architecture, which is the input language of the Larger IV silicon assembly environment. The final implementation is simulated and the results are compared to the simulation results generated by the Silage code. For the FPDA architecture, the approach to implementation and validation is to use VHDL, because VHDL allows commercial simulators and logic synthesis tools to be used in the behavioral synthesis design flow.

2.3 Conclusions

The previous research in behavioral synthesis was discussed. This research has provided the design flow used by behavioral synthesis tools. Previous research in behavioral synthesis for DSP application was also discussed. This research has focused on the development of custom, dedicated architectures and software synthesis for general purpose DSP microprocessors. Features of previous research employed by the project are also described.
CHAPTER 3

OVERVIEW OF THE SYNTHESIS SYSTEM

The Wizard synthesis system provides a behavioral synthesis design environment for application specific FPDA processors. In order to provide an open design environment, Wizard was designed in an object-oriented fashion, employing user extendible design libraries. An overview of the Wizard design flow is shown in Figure 3.1. Wizard supports three levels of design activity for FPDA systems: designing FPDA processors, designing DSP functions for the design library, and synthesis of DSP system for a FPDA target architecture.

In the first section of this chapter introduces the design of a new FPDA processor. The second section describes the development of application specific DSP functions for a FPDA processor. The last section presents the generation FPDA microcode for a DSP system.
3.1 FPDA processor design

One level of design activity is the design of a new FPDA processor. During the design of a new FPDA processor, the designer may create new, application specific
execution units. The characteristic of the execution unit, including the control signal information, is stored in the *execution unit library*.

Wizard's time-constrained behavioral synthesis engine can be used to design a semi-custom processor architecture. During the design of a new FPDA processor, the time-constrained synthesis engine could be used to profile a set of DSP algorithms. The profiles would indicate the resource needed to handle these algorithms. The resource requirements include:

- *processing elements*; the number and type of execution units contained in the processor.
- *buses*; the number of internal buses required to connect the processing elements.
- *registers*; the number of registers contained in the register file of each processing element.

When the resource constraints for the FPDA processor are set, they are stored in the *target architecture library*.

The *target architecture library* also stores operation mapping information. The operation map indicates which processing element is used for each operation supported by the FPDA. This information is used during module selection, to assign a execution unit type to each operation.
3.2 DSP function design

The second level of design activity is developing common DSP algorithms for an existing FPDA processor. These common algorithms, called functions in Wizard, provide commonly used DSP algorithms for the target application area. Once these functions are designed, they are stored in the function library.

There are several advantages to using the function library approach. First, the amount of time required to design a new system is reduce if the system can use previously designed functions. Second, the function library provides an added level of confidence, since the algorithms in the library should be rigorously tested. Third, if an algorithm is not present in the library, it can be added to promote design reuse. Finally, Wizard's multi-processor synthesis engine has been optimized to work with functions, as opposed to scheduling all of the operations contained in the functions.

3.3 DSP system design with FPDA's

The last design activity supported by Wizard is the construction of DSP systems using FPDA processors. The design starts with a behavioral description of the DSP system. The behavior of the system is described using a special-purpose HDL, VHDL Data Flow Language (VDFL). The VDFL language is based on a subset of VHDL, with extensions for more natural representation of DSP systems, derived from the DFL
language. A VDFL compiler is used to convert the VDFL description into VHDL simulation models and a SFG description of the system.

3.3.1 Behavioral specification and simulation

The VHDL simulation models allow floating-point and fixed-point simulations of the VDFL system to be performed. Using these simulation models, a VHDL simulation is used to verify the functionality of the behavioral algorithm description, as well as to check the effects of using fixed-point operations. Floating-point simulations are performed using the VHDL built-in type \texttt{REAL}, which corresponds to IEEE single precision floating-point. Fixed-point simulations are performed using a package which provides VHDL procedures for all VDFL operations.

Simulation of the floating-point and fixed-point VHDL models produces results which are used to analyze the VDFL specification and to verify the results of behavioral synthesis. By using an external data analysis program, such as Matlab, the results of the floating-point VHDL simulation are analyzed, thereby verifying the VDFL algorithm description. The data analysis program is also used to investigate fixed-point effects, by analyzing the results of the fixed-point simulation. The fixed-point VHDL simulation model is also used to check the results of behavioral synthesis, by comparing the results of the fixed-point simulation to the simulation results of the final RTL VHDL model.

The SFG model is used by Wizard throughout the synthesis process. The SFG contains nodes that represent operations in the VDFL description. SFG nodes are
connected by edges, which represent data flow in the SFG. During the synthesis process, intermediate results, such as start cycle, are back-annotated into the SFG database, for use by subsequent Wizard modules.

3.3.2 Behavioral synthesis engine

In the Wizard behavioral synthesis system, the behavioral synthesis engine is used to perform module selection, scheduling and binding. The synthesis engine supports single-processor and multi-processor FPDA architectures. The single-processor synthesis engine uses a resource-constrained synthesis algorithm to map the DSP algorithm to the FPDA processor. Wizard uses another resource-constrained synthesis algorithm to map a DSP algorithm to a predesigned system of FPDA processors. A time-constrained synthesis algorithm is also provided, which construct a multi-processor system based on a user supplied time constraint.

To begin the synthesis process, the designer specifies the target FPDA processor and the architecture. The first step of the synthesis process is module selection. Module selection is the process of binding VDFL operators to Execution Units (EU). Operation binding is performed using a mapping function. The mapping function is based on the EU's contained in each target architecture. By using an extendible operation map, Wizard allows new target architectures to be easily added to the design flow.

The next step of the synthesis process is scheduling and binding. In order to improve synthesis results, Wizard performs scheduling and binding simultaneously. During
scheduling, each operation is assigned to start on a specific cycle. During binding, each operation is assigned to a specific hardware instance.

The final step of the synthesis process is register allocation. Register allocation is the process of assigning every variable in the system to a specific register location. Lifetime analysis is used to minimize the number of registers used, by allowing variables with non-overlapping lifetimes to share the same register.

3.3.3 Code generation

Code generation is the process of converting the information produced by behavioral synthesis into a form that can be used for implementation. In the case of FPDA synthesis, the output of code generation is a microcode program. The code generator builds the microcode program based on the static schedule and resource binding produced by the synthesis engine. Control signal templates from the design library are used to set the control signal values for each execution unit and register file. These control signals are grouped into microcode instructions, which form the microcode program.

A VHDL behavioral model of the FPDA processor can then be used to validate the behavioral synthesis process. A behavioral simulation of the FPDA system is performed, using the microcode program generated by the behavioral synthesis tool. By comparing the results of this simulation to the results of the simulation of the behavioral specification, the synthesized system can be validated.
3.4 Conclusions

An overview of the Wizard behavioral synthesis system was discussed. The three development activities for FPDA systems were described. These activities include: FPDA processor design, DSP function design, and DSP system design using FPDA processors. Each of these design activities are discussed throughout this dissertation.
As behavioral synthesis tools move the design of digital systems to higher levels of abstraction, the design specification language should also move to higher levels of abstraction. A higher level specification medium allows the designer to describe the design in a more natural format. The VHDL Data Flow Language (VDFL) has been developed as the behavioral specification language for the Wizard behavioral synthesis system. VDFL has been developed to describe DSP systems in a more natural format. VDFL combines a subset of VHDL with extensions from Silage to create a powerful and flexible DSP specification language.

The first section of this chapter describes the motivation for creating the VDFL language. The second section provides a description of the VDFL language. The third section describes the language translators used to translate VDFL into VHDL. The forth section describes how VDFL is used for behavioral simulation.
4.1 Motivation for VDFL specification language

An example of this is seen in state diagram editors. Although VHDL is capable of directly describing finite state machines, some designers feel more comfortable using a state diagram format. For Digital Signal Processing (DSP) systems, Silage [8] and its commercial counterpart DFL [26] are more natural languages for design specification.

Instead of using assembly language or C to describe the DSP system, VDFL allows a system to be specified at a conceptually higher level. Since DSP algorithms are usually designed at a higher level, such as manipulating data streams with operators, VDFL provides a natural specification language for the designer. VDFL is not a simulation language, therefore language translators are used to convert VDFL to VHDL for behavioral simulation.

A specification should also be executable, in order to verify the accuracy of the description and to provide a “golden” model as synthesis tools move the specification towards implementation. VDFL code is translated to fixed-point and floating-point VHDL models to provide an executable specification. VDFL is also translated into a signal flow graph (SFG) for input into a behavioral synthesis system. As the design moves towards implementation, the VDFL model can be used to validate the lower-level VHDL models produced by behavioral synthesis and logic synthesis tools. In this section, the development of the VDFL language and the VDFL compiler used in the Wizard synthesis system is described.
The design of DSP algorithms usually takes place at a high level of abstraction, by considering the manipulation of data streams. A natural format for the description of DSP algorithms is the Signal Flow Graph (SFG) [27]. An SFG contains nodes, which represent operations, and edges, which represent data streams connecting these operations. The SFG does not specify how operations are bound to processors or how data streams are bound to storage. The advantage of omitting precise operation ordering information is that the behavioral synthesis system can find opportunities for parallel computation, and exploit this parallelism in order to reduce the amount of time required to execute the system for a specified target architecture.

An applicative language is the most natural way to represent SFG semantics in a textual format. A program written in an applicative language contains a set of equations, instead of assignments, and a set of values, instead of variables. The applicative language model is used by concurrent signal assignments in VHDL. The marriage of DSP system specification with VHDL is a natural solution, since they are both being used to describe the operation of a digital system at a higher level of abstraction.

VHDL has been shown to be an effective design environment for DSP [28]. VHDL can be used to model DSP algorithms at floating-point, fixed-point, or implementation levels. VHDL’s ability to model all levels of the design process supports a top-down design methodology for the design of DSP systems.

The motivation for developing VDFL is to provide a DSP specification language with the features of DFL which will take advantage of VHDL as a powerful simulation
language. There are several advantages to using a combination of these languages, as opposed to using VHDL or DFL alone, including:

- Using the DFL delay clause reduces the amount of programming overhead required to describe the same construct in VHDL.
- Using a subset of VHDL limits the number of VHDL constructs that must be supported by a behavioral synthesis tool.
- Using generics and the VHDL simulation engine provides more flexibility than is available from DFL alone.

VDFL is being developed as a language for a synthesis-driven design methodology. Therefore, the behavioral synthesis aspects, as well as the specification and simulation aspects of the language, must be considered. Designing the language in this manner results in a powerful specification and simulation language, which can be used to produce high quality behavioral synthesis results.

4.2 VDFL language description

The syntax of VDFL closely matches the syntax of VHDL [6]. Since the VHDL language is designed to address a large number of applications at varying levels of abstraction, VDFL accepts only a limited number of VHDL constructs. VDFL introduces some extensions to VHDL, which allow a more concise representation of DSP algorithms, and VDFL also defines a new type specification which is used to represent fixed-point values.
Adopting the VHDL syntax simplifies the translation of VDFL to VHDL, because many of the program constructs will not need to be changed during translation. Matching the syntax of VHDL also allows a VDFL parser to be easily created. Modifying an existing VHDL parser to handle the extensions from DFL and the limitations of a VHDL subset will produce a VDFL parser which can be used for language translation. In this section, a description of the VDFL language is provided.

4.2.1 Lexical elements

Lexical elements in VDFL correspond directly to lexical elements in the VHDL language, with some additional operators. A summary of the VDFL lexical elements is shown in Table 4.1. By using the same lexical element definitions as VHDL, all identifiers, keywords, constants, and operators defined in a VDFL specification will be valid in the VHDL simulation models.

4.2.2 Data types

Signals defined in a VDFL specification must use one of the types shown in Table 4.2. These types correspond to the signal types defined in DFL. During source translation, the VDFL types are transformed into VHDL types for simulation.

For DFL float types, m corresponds to the mantissa length of a signal and e corresponds to its exponent length. Although DFL supports the specification of the
mantissa and exponent length for floating-point numbers, the support for simulation of this feature is implementation dependent. Currently, VDFL does not support specification of the mantissa and exponent length for floating-point numbers, since it is not supported by VHDL simulation. This feature could be added to VDFL and supported using VHDL floating-point operations [29].

In DFL fixed-point types, \( w \) corresponds to the wordlength of the signal and \( f \) corresponds to its fraction length. Fixed point signal definitions in VDFL are translated into an array of \text{STD\_LOGIC} in VHDL. By using type \text{integer} to define the array range, instead of type \text{natural}, the location of the binary point is preserved. An example fixed-point signal declaration, its bit representation, and corresponding VHDL signal declaration are shown in Figure 4.1.

Arrays of signals can also be declared in VDFL. In order to declare an array of signals, an array range is appended to the type indication. The VDFL translator generates any intermediate array types required by the VHDL simulation models.
<table>
<thead>
<tr>
<th>Lexical Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifiers</td>
<td>VHDL Identifiers (case insensitive)</td>
</tr>
<tr>
<td>Keywords</td>
<td>VHDL Keywords, VDFL Types</td>
</tr>
<tr>
<td>Constants</td>
<td>VHDL Literals</td>
</tr>
<tr>
<td>Operators</td>
<td>VHDL Operators, VDFL Operators</td>
</tr>
</tbody>
</table>

Table 4.1: VDFL lexical elements

<table>
<thead>
<tr>
<th>DFL</th>
<th>VDFL</th>
</tr>
</thead>
<tbody>
<tr>
<td>float&lt;mantissa, exponent&gt;</td>
<td>real</td>
</tr>
<tr>
<td>fix&lt;width,fraction&gt;</td>
<td>fix&lt;width, fraction&gt;</td>
</tr>
<tr>
<td>int&lt;width&gt;</td>
<td>int&lt;width&gt;</td>
</tr>
<tr>
<td>unsigned&lt;width, fraction&gt;</td>
<td>ufix&lt;width, fraction&gt;</td>
</tr>
<tr>
<td>unsigned_int&lt;width&gt;</td>
<td>uint&lt;width&gt;</td>
</tr>
<tr>
<td>bool</td>
<td>bool</td>
</tr>
</tbody>
</table>

Table 4.2: Translation of DFL data type to VDFL data types
Fixed-point signal representation:

$\text{fix}<10, 7>$

Integer signal representation:

$\text{int}<10>$

 Unsigned fixed-point signal representation:

$\text{ufix}<10, 7>$

 Unsigned integer signal representation:

$\text{uint}<10>$

Figure 4.1: VDFL signal representations
### Table 4.3: Fixed-point operations in VDFL

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Left Operand</th>
<th>Right Operand</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>Bitwise negation</td>
<td>fix(&lt;w, f&gt;)</td>
<td>fix(&lt;w, f&gt;)</td>
<td>fix(&lt;w, f&gt;)</td>
</tr>
<tr>
<td>+, -</td>
<td>Unary plus</td>
<td>fix(&lt;w, f&gt;)</td>
<td>fix(&lt;w, f&gt;)</td>
<td>fix(&lt;w, f&gt;)</td>
</tr>
<tr>
<td></td>
<td>Unary minus</td>
<td>fix(&lt;w, f&gt;)</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>fix(&lt;w1, f1&gt;)</td>
<td>fix(&lt;w2, f2&gt;)</td>
<td>fix(&lt;w1+w2, f1+f2&gt;)</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>fix(&lt;w, f&gt;)</td>
<td>fix(&lt;w, f&gt;)</td>
<td>fix(&lt;w, f&gt;)</td>
</tr>
<tr>
<td>+, -</td>
<td>Add</td>
<td>fix(&lt;w, f&gt;)</td>
<td>fix(&lt;w, f&gt;)</td>
<td>fix(&lt;w, f&gt;)</td>
</tr>
<tr>
<td></td>
<td>Subtract</td>
<td>fix(&lt;w, f&gt;)</td>
<td>fix(&lt;w, f&gt;)</td>
<td>fix(&lt;w, f&gt;)</td>
</tr>
<tr>
<td>&gt;&gt;, &lt;&lt;</td>
<td>Shift right</td>
<td>fix(&lt;w, f&gt;)</td>
<td>uint(&lt;w2&gt;)</td>
<td>fix(&lt;w1, f&gt;)</td>
</tr>
<tr>
<td></td>
<td>Shift left</td>
<td>uint(&lt;w1, f&gt;)</td>
<td>uint(&lt;w2&gt;)</td>
<td>uint(&lt;w1, f&gt;)</td>
</tr>
<tr>
<td>&lt;, &gt;, &lt;=</td>
<td>Relational operators</td>
<td>fix(&lt;w1, f1&gt;)</td>
<td>fix(&lt;w2, f2&gt;)</td>
<td>BOOL</td>
</tr>
<tr>
<td>&gt;=, =, /=</td>
<td>Operators</td>
<td>fix(&lt;w1, f1&gt;)</td>
<td>fix(&lt;w2, f2&gt;)</td>
<td>BOOL</td>
</tr>
<tr>
<td>AND, OR, XOR</td>
<td>Logical operators</td>
<td>fix(&lt;w, f&gt;)</td>
<td>fix(&lt;w, f&gt;)</td>
<td>fix(&lt;w, f&gt;)</td>
</tr>
</tbody>
</table>

4.2.3 Operators

VDFL provides operators common to most programming languages. Unary operations, which require one operand, and binary operations, which require two operands, are provided. For fixed-point numbers, the result type of an operation is a function of the operand types and the operator. Operand type requirements and implicit type casting of results have been adopted from DFL, as shown in Table 4.3.
VDFL also provides constructs unique to the description of DSP systems. The time delay clause, @, brings the concept of samples into the specification language. For example, the VDFL sample delay statement, shown in Figure 4.2, assigns to y the value of x, delayed by three samples. The time delay clause simplifies the specification of DSP algorithms compared to writing VHDL code to provide the same functionality, which is also shown in Figure 4.2.

There are several advantages to providing the time delay clause. The most obvious advantage is that the amount of programming required to describe a sample delay is reduced. Another important advantage is that the specification is more precise. In the example in Figure 4.2, there is no question what is meant when using the delay clause, but
the VHDL description is ambiguous. The more precise representation of the sample delay benefits designers and the synthesis tools. The designers benefit from the precise representation because it provides self-documenting code. A synthesis tool benefits from the precise representation because it does not have to figure out what the designers want to describe. Since VHDL is a simulation language, there is usually more than one way to describe a desired operation. By providing one unique method of describe sample delays, the synthesis tool can easily recognize this construct and provide an optimal solution.

4.2.4 Expressions

An expression is a formula that defines the computation of a value. Expressions are composed of primaries with optional operators. A primary has a value and a type. There are three types of expressions defined in VDFL: unary, binary, and conditional. Unary expressions are expressions that take only one operand. Binary expressions are expression that take two operands. The operators for these expressions are shown in Table 4.3.

Conditional expressions define a signal value based on the result of condition expressions. This construct directly maps to the conditional signal assignment statement in VHDL.
4.2.5 Miscellaneous language constructs

Two other VHDL constructs have been adopted by the VDFL language: generics and generate statements. The generic construct provides a mechanism to pass static information to a VDFL model. By using generics, one VDFL specification can be used to model a large class of DSP algorithms. The generate statement allows iterative or conditional elaboration of concurrent statements. Generate statements provide a concise way to represent a large number of signal assignments.

4.3 Translation of VDFL to VHDL for behavioral simulation

Language translators are used to translate the VDFL specification into floating-point and fixed-point VHDL models for behavioral simulation. A translator is also used to convert the VDFL specification into an SFG for input to the behavioral synthesis process. The PCCTS [30] language translation tools are used to generate the C++ language translation programs. In this section, the VDFL language translators are described.

PCCTS provides a set of tools that is well suited for source-to-source translation. The tools include ANTLR, which is a top-down parser generator, and SORCERER, which is a tree-parser generator. There are several advantages in choosing ANTLR over YACC for language translation. ANTLR accepts grammar definitions in EBNF format, provides automatic error recovery and reporting, and automatically generates an Abstract Syntax Tree (AST). YACC does not accept EBNF format, does not provide automatic
error recovery or reporting, and does not generate an AST. SORCERER generates a tree-walker, which is used to parse the AST produced by ANTLR. By embedding actions in the AST grammar, the tree-walker is used to translate the AST to the target language.

An outline of the language translation process used by Wizard is shown in Figure 4.3. The VDFL grammar description is common to all language translators. Different tree-parser specifications are used to translate the VDFL AST into the desired target language. The VDFL parser provides syntactic error reporting. Semantic error reporting is also provided, through the use of a symbol table. A symbol table keeps track of all declared identifiers, their types, and values. During parsing, certain identifiers are checked for presence in the symbol table. If the identifier is not present, this indicates a semantic error. The symbol table is also used by the AST parsers during target language generation. The AST parsers use the type and value information to generate the new source file.
Figure 4.3: VDFL translation
4.4 Behavioral simulation

Behavioral simulations are an important step in the design flow for two reasons. First, simulation results can be used to verify that the VDFL specification meets the expectations of the designer. Second, behavioral simulation results can be used to validate the synthesis results at later stages of the synthesis process. Wizard provides two simulation models for behavioral simulation: floating-point and fixed-point. The results of the VHDL behavioral simulation are stored to a file and processed by an external application. Since VHDL is used to simulate VDFL models, register transfer level VHDL models produced by behavioral synthesis and gate level VHDL models produced by logic synthesis can be directly validated using simulation. In this section, the generation of the floating-point and fixed-point models is described. Application specific processing of simulation results is also discussed.

4.4.1 Floating-point simulation

The floating-point model produced by the VDFL language translator is generated by replacing all fixed-point types with floating-point types. Simulations using the floating-point models allow the VDFL specification to be verified at an algorithmic level. For example, the impulse response of a digital filter can be generated and compared to the expected impulse response.
The precision of the floating-point simulation is dependent on the precision of the VHDL type \texttt{REAL} provided by the VHDL simulator. DFL provides the ability to specify the precision of floating-point data types, but the DFL simulator does not support this feature. The VDFL language could be expanded, if the capability to control the precision of floating-point were desired. A method to control the precision of floating-point operations in VHDL, which is described in [29], could be used to support this feature in a VHDL simulation environment.

4.4.2 Fixed-point simulation

The fixed-point model produced by the VDFL language translator is generated by replacing all fixed-point type declarations with valid VHDL type declarations. Fixed-point signals for intermediate results are also generated during language translation. Fixed-point simulations allow the finite wordlength effects to be investigated at an early stage in the design process. For example, a frequency response plot can give the designer insight on how fixed-point wordlength affects the operation of the system, and adjustments in the wordlength can be made to meet the design requirements.

The size of fixed-point words used in the VHDL simulation is unlimited. A fixed-point package is provided to implement all fixed-point operations described in Table 4.3. The operations in the fixed-point package are implemented using the IEEE \texttt{NUMERIC_STD} package [31], which provides arithmetic operations for \texttt{STD_LOGIC_VECTOR} signals.
When simulating an algorithm that will be mapped to a FPDA processor, the fixed-point precision of VDFL operations must match the precision provided by the processor. In this case, the VDFL operation result precision might not match the precision shown in Table 4.3. Therefore, the VDFL translator must know the target FPDA processor, so that the correct precision will be used in the behavioral simulation.

4.4.3 Simulation result analysis

The results of the VHDL simulations are written to a file, which can be processed by external data analysis tools. Since the type of simulation data may come from a wide variety of formats, the result analysis tool will depend on the application area. For digital filter design, a frequency response plot would be a useful data analysis tool. For image processing applications, a digital image viewer would be a useful analysis tool. Once a method for viewing the image result is developed, it can be to view the simulation results of lower-level VHDL models, as well as new applications in the same area.

4.5 Conclusions

The VDFL behavioral specification language was discussed. The VDFL language is used to provide a behavioral specification to the Wizard synthesis system. VDFL is converted to VHDL for behavioral simulations and to an SFG for input into the synthesis process.
CHAPTER 5

FPDA TARGET ARCHITECTURE

This research introduces the FPDA processor architecture to the behavioral synthesis design flow. This architecture supports the design of application specific, programmable DSP systems using single processor and multi-processor configurations. A generic FPDA has a Very Long Instruction Word (VLIW) architecture, which contains processing elements with distributed memories connected by a network of buses. The system is controlled by a microcode controller. Extensions to the FPDA architecture have been added to support the design of multi-processor systems.

In the first section of this chapter, the generic FPDA architecture is introduced. In the second section, application profiling is introduced. In the third section, the FPDA architecture extensions to support multi-processor designs are described.
5.1 Generic FPDA architecture

The architecture developed for the FPDA processors is shown in Figure 5.1. This architecture is similar to general purpose microprocessor architectures. Unlike a general purpose microprocessor, the FPDA is a distributed memory architecture that supports
multiple register transfers per cycle, which is necessary for real-time DSP algorithms. The major components of this architecture include: the processing elements, a communication network, and the system controller. Each processing element contains a register file and an execution unit, as shown in Figure 5.2. In this section, the components of the FPDA architecture are described.

5.1.1 Execution units

Execution units provide all data manipulation in the FPDA architecture. The execution units can be simple or complex functional units, depending on the system application. This is one feature of the architecture that allows Wizard to provide an application specific programmable FPDA processor. By designing the behavioral synthesis tool in an object-oriented fashion [32-34], the synthesis algorithms are implemented independent of the execution unit function. Furthermore, by using a user extendible library, new execution units can be added to the design flow without changing or recompiling the synthesis tool.
In the synthesis system, an execution unit is modeled as a black box, as shown in Figure 5.3. Each execution unit has two input ports, one output port, and a set (possibly empty) of control signals. This execution unit model supports single-cycle and multi-cycle operations, as well as pipelined and non-pipelined execution units. This model also allows
Wizard to seamlessly support the addition of user designer execution units to the synthesis flow.

The execution unit library is used to hold descriptions of all execution units supported by the synthesis tool. An entry in the execution unit library is called an EU Record. An example EU record is shown in Figure 5.4. The EU record contains the following information:

- CONTROL_SIGNALS: This entry gives a list of all control signals used by the execution unit. If this entry is not present, then the execution unit does not require and control signals. The width of each control signal is also indicated.
• OPERATION: The operation entry contains a list of operation records. An operation record is provided for each operation supported by the execution unit.

The operation record contains information specific to a particular operation. All operation records contain the following information:

• CYCLES: This entry indicates the number of cycles required to execute the operation.

• OUTPUT_READY: This entry indicates when the output of an operation is produced, relative to the start of the operation. For example, a value of "1" means that a result is produced on the first cycle of the operation. For operation that produce more than one result, this entry will contain a list of values.

• CONTROL: The control entry contains a list of control signal records. A control record is used to indicate the values that must be applied to the control signals for the specified operation.
The control record is used by the code generator to produce the microcode program for the system controller. Each control record contains the following information:

- **START**: This entry indicates the beginning cycle of the operation in which this record is valid.

- **STOP**: This entry indicates the last cycle of operation in which this record is valid.
• VALUES: This entry is a list of values that must be applied to the control signals during the range of operation cycles indicated by [START, STOP]. Each entry in the value list directly corresponds to entries in the CONTROL_SIGNAL list of the EU record. A value of "-1" indicates a don't care value for the control signal during the cycle range.

All value entries in the EU record that are enclosed by quotation marks are represented by expressions. Using expressions allows implementation dependent information, such as wordlength or coefficient values, to be used by the execution unit library. The only requirement of using these variables is that they must be defined in the signal flow graph. The grammar used by the execution unit library is provided in APPENDIX.

The number and type of execution units contained on a processor is a variable of the FPDA architecture. Changing the number and type of execution units is another feature of the architecture that allows Wizard to provide application specific programmable FPDA processors. The execution unit requirements should be set based on the target application requirements. To quantify these requirements, application profiling will be used

5.1.2 Register file and connection network

Register files are used for data storage in the FPDA architecture. These small, distributed memories provide data to the execution units in each processing element. There are several advantages to using this memory architecture. First, the register files are
small compared to a large shared memory. Second, the buses between the register file and execution unit can be short, since the register file and its corresponding execution unit can be located physically close to each other. Third, the number of I/O ports required by a register file is fewer than one large shared register file.

A disadvantage of using distributed memories is the potential for duplication of data. If more than one processing element needs to use the same data, storage will be wasted. In a single, shared register file architecture, no storage would be wasted, because data is would not be duplicated. This issue is addressed by the Wizard synthesis engine, which tries to bind operations that use the same data to the same processing element. Even when the compiler fails to optimize the operation binding, performance and area gains from using the distributed memory architecture outweigh the disadvantage of duplicate storage.
A block diagram of a generic register file is shown in Figure 5.5. Each register file contains two read ports and at least one write port. The size of a register file indicates the number of registers it contains. This number is a variable of the FPDA architecture. Since the FPDA does not support external memory, all storage requirements of the processor must be handled by the register files. Application profiling will be used to set the register file size requirements for the target application.
The number of write ports contained in each register file is another variable of the FPDA architecture. In the FPDA architecture, this number constrained to be equal to the number of internal buses used by the connection network, which is shown in Figure 5.6. This requirement allows a result on the connection network to be written to any register file on the FPDA. Given this constraint, the task of the synthesis engine is schedule operations so that the number of results produced in any cycle is less than or equal to the number of internal buses.

Since the number of results produced every cycle is limited by the number of internal buses, increasing the number of internal buses increases the total system performance. Increasing the number of internal buses also increases the complexity of the system. Therefore, the number of internal buses should be set to match the processing requirements of the target application. The factors to consider when setting the number of internal buses include: the number of processing elements on the chip and the average results per cycle for typical algorithms from the application area.

For maximum throughput, the total number of internal buses is the number of processing elements plus the number of input ports. This connection network would allow each processing element to produce a result every cycle. It would also allow the FPDA processor to accept an input on all input ports for every cycle. For most application areas, a full connection network is not required.

The average results per cycle (RPC) is a measurement of the average number of results produced during each cycle of an algorithm specific to the application area. The RPC is a function of the number of processing elements used, the number of cycles
required for each operation, and the parallelism of the DSP algorithm. Clearly, the RPC is application dependent and algorithm dependent. Therefore, application profiling will be used to determine the internal bus requirements.

Figure 5.6: Internal connection network for a single-processor FPDA
5.1.3 System controller

The system controller used by Wizard is a VLIW microcode programmable controller. Since this target architecture does not include looping or control structures, the system controller does not receive input from the execution units. The output of the system controller is the set of control signals required by the processing elements and the connection network. The implementation style of the microcode programmable controller is not specified by the FPDA architecture.

One possible implementation of the microcode programmable system controller is shown in Figure 5.7. This controller uses horizontal microcode to provide each processing element with their control signals. The address of the next microinstruction is included in every microinstruction. Including the next microinstruction in each control word reduces the amount of hardware needed to compute the address of the next microinstruction.

The size of the microcode memory is another variable of the FPDA architecture. The length of each word in the memory will be fixed by choices made for other variables in the FPDA architecture, namely processing elements and internal buses, but the number of words in the memory is application dependent. The number of words can be set based on the FDPA system clock frequency and the expected sample frequency of the target application. The maximum number of words required in the microprogram memory is given by the following formula:

$$\text{words} = \left\lceil \frac{\text{clock\_frequency}}{\text{sample\_frequency}} \right\rceil$$
The memory size will usually be adjusted to the next highest power of two, to take advantage of the memory structure. The adjusted number of words for the microprogram memory is given by this formula:

$$\text{adjusted\_words} = 2^{\left\lceil \frac{\ln(\text{clock\_frequency})}{\ln(\text{sample\_frequency})} \right\rceil}$$

The extra microprogram memory is useful for two reasons. First, the extra memory words can be used for initialization instructions. Initialization instructions might be required by a
DSP algorithm to load coefficients into the FPDA. By using microcode to load these coefficients into the registers files, the need for extra internal hardware and instructions is eliminated. Additional microprogram memory is also useful if the system clock frequency is increased or if the frequency range of the target application is decreased. If either or both of these frequencies change, the maximum memory requirements of the microcode program will change. Therefore, the number of words in the microprogram memory should be set based on:

- System clock frequency
- Application sample frequency
- Initialization instruction requirements

The potential for change in any of these parameters should also be weighed in the computation of the memory size.

5.2 Application profiling

Application profiling is used during the design of an FPDA processor in order to match the functionality of the processor to the application area. Application profiling is the process of gathering resource requirements for a set of DSP algorithms in the target application area. The resource requirements are determined based on the maximum desired sample rate of the algorithm and the system clock frequency of the FPDA processor. These resource requirements include: the number and type of processing elements, the number of internal buses, and the number of registers in the register files.
In order to support application profiling, Wizard provides a time-constrained scheduler for a custom FPDA architecture. This scheduler minimizes the resources requirements for a FPDA processor, given a constraint on the number of clock cycles to execute the algorithm. Therefore, synthesizing the application specific set of DSP algorithms using the custom FPDA synthesis engine gives the minimum resource requirements for the desired functionality.

5.3 Multi-processor FPDA architecture

The multi-processor FPDA architecture is optimized to allow communication between FPDA processors. The multi-process topology supported is shown in Figure 5.8. The variables of a multi-processor FPDA architecture are the number of input ports for each processor, the number of external buses, and the number of processors. The number of input ports and the number of external buses must be equal. This constraint is analogous to the constraint of setting the number of register file write ports equal to the number of internal buses. Setting the number of input ports equal to the number of external buses insures that a result on the external bus can be written to any processor in the system. Since the number of input ports will be fixed by the FPDA processor, the number of external buses must be less than or equal to the number of input ports. The number of processors is not constrained by the FPDA architecture.
A multi-processor FPDA processor differs from a single-processor FPDA in the construction of the internal connection network. The connection network for a multi-processing enabled FPDA processor is shown in Figure 5.9. All processing elements in a multi-processor FPDA must include an input for each processor input port. This allows a result on the external bus to be written to any processing element in the system. A disadvantage of this approach is that the synthesis engine must insure that the number of results passed between processors is less than or equal to the number of external buses. This becomes a difficult partitioning problem, which must optimize processing element
utilization, register requirements, and inter-processor communication. This problem is solved in Wizard using another approach.

For multi-processor FPDA system, Wizard schedules functions, as opposed to operations. A function is a commonly used DSP block, which can be scheduled on one
FPDA processor. By scheduling a function on one processor, most of the communication is localized to the internal buses. The synthesis engine then insures that the number of function results produced in each cycle, for the entire system, is less than or equal to the number of input ports. A function result is defined as a output of the function that is used by another function. The number of results produced by a function will usually be much smaller than the execution length of the function. This is analogous to the results per cycle measurement which is used to set the number of internal buses. For functions that produce less than one operation per cycle, the RPC will always be less than one. For many functions, the RPC will be much less than one. Therefore, the number of input ports on an FPDA processor can be set to one, in order to reduce the processor pin count. This scheduling constraint supports the usage of shared input and output ports, allowing the further reduction of processor pins.

5.4 Conclusions

The FPDA processor architecture was introduced. This architecture provides a flexible method to develop microcode programmable, application specific signal processors. Multi-processor extensions to the FPDA architecture were described. The constraint of using DSP functions for multi-processor synthesis was introduced. Designing a DSP system using functions provides a natural partitioning of the system. Using this method for the behavioral synthesis of multi-processor systems provides very good results in a very short time, without trying to solve a difficult partitioning problem.
CHAPTER 6

BEHAVIORAL SYNTHESIS FOR THE FPDA ARCHITECTURE

The behavioral synthesis engine is used to statically schedule and bind each operation of a DSP algorithm. In order to provide an extendible synthesis system, Wizard uses design libraries to provide information about the target FPDA. Wizard performs module selection, scheduling, and binding in the synthesis engine. Module selection is performed based on the operation mapping defined by the target FPDA processor. A list-based scheduling algorithm is used to perform simultaneous scheduling and binding. These algorithms provide very good results with low computational complexity. An algorithm developed to solve the register sharing problem is used after scheduling is complete.

In the first section of this chapter describes the design libraries used by the synthesis engine. The second section provides the generic synthesis algorithms used by the synthesis engine. The third section describes how the generic algorithms are applied for each target architecture.
6.1 Design libraries

In order to support the addition of new FPDA targets to the synthesis engine, design libraries are used to provide processor resource information. The execution unit library, described in Section 5.1.1, is one of the design libraries. The two other libraries used by Wizard are the target architecture library and the function library. The usage and format of both of these libraries are described in this section.

6.1.1 Target architecture library

The target architecture library is used by Wizard to read the resource constraints of the target FPDA processor. These constraints include: the number and type of processing elements, the number of internal buses, and the number of registers available in each register file. Operation mapping, which is used to assign the operations in the DSP algorithm to processing elements on the FPDA processor, is also included in the target architecture library. An entry in the target architecture library is called a target record.

An example target record is shown in Figure 6.1. Each target record contains the following information:

- TARGET: This entry gives the name of the FPDA processor.
- PE: Each PE entry indicates a processing element type and the number of processing elements of that type on available on the processor. Entries for the number of input and output ports are also included in this list.
Figure 6.1: A target record from the target architecture library

- **BUS**: This entry indicates the number of internal buses used in the connection network.
- **REG**: This entry indicates the number of registers available in the register file of each processing element.
- **MAP**: The mapping entry indicates which processing element type must be used for operations in the DSP algorithm.

These entries are stored in a file, which is read by the synthes system. Using an external file in this way allows Wizard to be expanded as new FPDA processors are developed.
Figure 6.2: A function record from the function library

6.1.2 Function library

The function library is used by Wizard to store application specific DSP algorithms. Each function stored in the library is prescheduled for a specific FPDA processor. As described in Section ?, functions allow Wizard to provide high quality multi-processor designs by providing a natural partitioning of the DSP system being designed. An entry in the function library will be called a function record.
An example function record is shown in Figure 6.2. Each function record contains the following information:

- **FUNCTION**: This entry indicates the target FPDA processor and the name of the DSP function.
- **CYCLES**: This entry indicates the number of cycles required to execute the function.
- **OUTPUT READY**: This entry indicates when the output of an operation is produced, relative to the start of the operation. For operation that produce more than one result, this entry will contain a list of values. This entry is analogous to the OUTPUT READY entry in the execution unit library.
- **SFG_FILE**: This entry provides the file name of the prescheduled SFG file for the function. This file is used during code generation to create the microcode program for the system.
- **PE USAGE**: This entry contains a list of usage records. A usage record indicates the processing element requirements for the function. The usage record is used during synthesis to determine when a FPDA processor has the resources available to schedule a function. Each usage record contains the following information:
  - **PE_ID**: This entry indicates a unique identifier for the processing element, which is used in the prescheduled SFG file.
  - **EU**: This entry indicates the execution unit type of the processing element.
• TEMP: This entry indicates the amount of temporary storage required on this processing element. Temporary storage is defined as the number of registers used during the lifetime of the function. These registers can be used for other purposes outside the lifetime of this function.

• PERM: This entry indicates the amount of permanent storage required on this processing element. Permanent storage is defined as the number of registers used during the entire system lifetime. Permanent storage is used to keep values from the previous sample period, such as delay operations. These registers cannot be used for other purposes outside the lifetime of this function.

• BUS_USAGE: This entry contains a list of values that indicate when the processing element requires the use of an internal bus.

These entries are stored in a file which is read by the synthesis system. Using an external file in this way allows Wizard to be expanded as new DSP algorithms are created and as new FPDA processors are developed. The function library also allows users to add their own DSP algorithms to the behavioral synthesis design flow.

6.2 Synthesis algorithms

The synthesis algorithms used by Wizard have been optimized to work with the FPDA architecture. These algorithms include: module selection, simultaneous scheduling and binding, and register binding. The operation of each of these algorithms is described in this section.
In order to synthesis results, Wizard performs scheduling and binding simultaneously. Concurrent scheduling and binding allows the effects on processing element, bus, and register usage to be considered simultaneously, thereby improving the behavioral synthesis results. When hardware constraints are not set, Wizard performs hardware allocation in parallel with scheduling and binding. Wizard provides a technique to improve hardware allocation results for list-based scheduling.

There are two categories of scheduling algorithms: time-constrained and resource-constrained. Wizard solves both of these scheduling problems with a list-based scheduling algorithm. There are several advantages to using a list based scheduling algorithm. First, a list based scheduling algorithm provides good results in a short execution time. Second, since the computational complexity of a list scheduling algorithm is low, large designs can be handled. Third, list based algorithms can easily be customized to match the target architecture of the synthesis system.

After the scheduling is complete, a register binding algorithm uses lifetime analysis to reduce the number of registers used by the system. Wizard uses an algorithm based on the left-edge algorithm [35] to solve the register sharing problem. This algorithm has been developed to solve the problem of binding registers with cyclic lifetimes, such as delay registers in a DSP algorithm.
6.2.1 Module selection

Module selection is the process of assigning each operation in the DSP algorithm to a processing on the FPDA processor. Module selection is performed by mapping an operation to an execution unit type. When synthesizing for a FPDA processor, module selection reads the operation mapping entries from the target architecture library and assigns each operation the corresponding execution unit type. When developing a new FPDA processor, module selection allows the designer to compare the usage of different execution units in the FPDA processor.

One advantage of using this approach is that module selection can be controlled by the designer. When designing an application specific processor, the designer usually has a good idea about what type of execution unit they want the synthesis tool to use. This is especially true when an application specific execution unit has been added to the execution unit library. By giving the designer control of module selection, the need to "coerce" the synthesis tool to choose the desired execution unit is eliminated. Another advantage of the target architecture library is that it promotes design reuse. Once a set of application specific executions units has been developed, they can be reused by the synthesis system on demand.

A limitation of this approach is that we complicate design space exploration. An advantage of the high-level synthesis methodology is the ability to explore different design implementations, since the behavioral description is implementation independent. For synthesis tools that provide automated module selection, design space exploration can also
be automated. In order to explore different design implementations in Wizard, the designer must create different target architecture definitions. Once these architecture definitions are created, Wizard can be used to automatically explore different design implementations.

6.2.2 Time-constrained scheduling

Given a desired execution time, the time-constrained scheduler minimizes the number of hardware resources required to execute an DSP algorithm. After scheduling a time-constrained algorithm, Wizard reports the number of resources required to implement the design. Time-constrained scheduling is used for application profiling during the creation of a new FPDA processor. By scheduling a set of algorithms that the new FPDA processor will be expected to handle, the resource constraints for the new processor can be determined. Time-constrained scheduling is also used for multi-processor synthesis. In time-constrained multi-processor scheduling, Wizard minimizes the number of processors required to execute a DSP algorithm in the specified execution time.
TimeConstrainedSchedule(SFG) {
    Compute ALAP schedule;

    Allocate minimum hardware resources;

    // restart scheduling if new resources are added
    restart = true;
    while(restart == true) {
        restart = false;

        Clear previous schedule;
        cycle = 0;

        // schedule all operations
        while(unscheduled ops AND restart == false) {

            // find nodes that are ready to be scheduled
            Determine candidate ops;

            Schedule ops with zero slack, add resources if needed;

            if(new resources were added) {
                restart = true;
            }

            Schedule ops that require no additional resources;

            cycle = cycle + 1;
        }
    }
}

Figure 6.3: Time constrained scheduling algorithm used by Wizard.
The time-constrained scheduling algorithm used by Wizard is shown in Figure 6.3. The first step of the algorithm is to compute the as late as possible (ALAP) schedule for the SFG. The ALAP start time is used during scheduling to compute the slack of an operation, which is defined as the difference between the ALAP start time and the schedule step under consideration.

The next step of the algorithm is to perform the initial hardware allocation. The initial hardware allocation is set to the lower bound of execution units and buses required to schedule the algorithm in the specified iteration period. The lower bound on the number of execution units is defined as:

\[
EU_i = \left\lfloor \frac{op_i \times cycles_i}{T} \right\rfloor
\]

Where \( op_i \) is the number of operations of type \( i \), \( cycles_i \) is the number of cycles required to execute operation \( i \), and \( T \) is the total number of cycles available. The lower bound on the number of buses needed to connect the execution units is defined as:

\[
buses = \left\lfloor \frac{results}{T} \right\rfloor
\]

Where \( results \) is the total number of results produced by all operations in the SFG and \( T \) is the total number of cycles available. This definition encompasses operations that generate more than one result.

After the initial hardware allocation is complete, the nodes are scheduled and bound to the available hardware. First, the scheduler constructs a set of candidate operations. An operation is selected as a candidate for scheduling if all of its predecessors
have completed their operations. The candidate operations are stored in a priority list, where the operations with lower slack have higher priority.

At this point, all candidate operations with zero slack are scheduled. An operation with zero slack indicates that the cycle under consideration is the ALAP start time for that operation. If the operation cannot be scheduled using the currently allocated resources, new hardware is allocated by the scheduler. Finally, operations in the candidate list that require no additional resources are scheduled.

Wizard improves on the results of the list-based scheduling algorithm presented in [11] by restarting the scheduling algorithm when new resources are allocated. By restarting scheduling after allocating new resources, previously scheduled operations can utilize resources that were not available when they were first scheduled. This can greatly affect the amount of hardware required to schedule a time-constrained algorithm, as shown by the scheduling results in Table 6.1. This table shows the results of scheduling a third-order IIR filter with and without the restart mechanism for several time constraints.

Another benefit of the restart algorithm is that the number of cycles required to execute the algorithm will be minimized based on the amount of hardware required to meet the time constraint. For example, a time constraint of 16 cycles requires one multiplier, one adder, and two buses. The restart scheduling algorithm reduces the number of cycles required to execute the algorithm to 11.
Table 6.1: Comparison of scheduling algorithms for a 3rd order IIR filter.

<table>
<thead>
<tr>
<th>Time Constraint</th>
<th>Without Restart</th>
<th>With Restart</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mult</td>
<td>Add</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

6.2.3 Resource-constrained scheduling

Resource-constrained scheduling minimizes execution time for a DSP algorithm given a fixed amount of hardware resources. Wizard uses resource-constrained scheduling to synthesize the microcode program for single-processor and multi-processor FPDA systems. Resource-constrained, multi-processor FPDA system are systems with a fixed number of FPDA processors. The FPDA multi-processor architecture, described in Section 5.3, defines how the processors are connected. Therefore, the resource-constrained scheduler can be used to generate microcode for a predesigned multi-
ResourceConstrainedSchedule(SFG) {
    compute ASAP and ALAP schedules;
    read resource constraints from library;
    cycle = 0;
    // schedule all operations
    while(unscheduled ops) {
        // find nodes that are ready to be scheduled
        Determine candidate ops;
        Schedule ops that fit on the available resources;
        cycle = cycle + 1;
    }
}

Figure 6.4: Resource constrained scheduling algorithm used by Wizard.

processor system. The single-processor scheduler is a special case of the multi-processor scheduler, where the number of processors is set to one, and the scheduler does not have to optimize the schedule for multi-processor communication.

The resource-constrained scheduling algorithm used by Wizard, shown in Figure 6.4, is an extension of Hu's algorithm [36]. The first step of the algorithm is to compute the as soon as possible (ASAP) and as late as possible (ALAP) schedule for the SFG. The ASAP schedule is used to determine the critical path length in the SFG, which is used as
the time-constraint for computing the ALAP schedule. The ALAP start time is used during scheduling to compute the slack of an operation.

The hardware allocation is read from the target architecture library. The target architecture defines the resources available on a FPDA processor. Therefore, hardware allocation is fixed, and is not changed by the resource-constrained scheduler. The remainder of the resource-constrained scheduling algorithm is similar to the time-constrained algorithm previously described. Specifically, a list of candidate operations is constructed, and operations that fit on the available resources are scheduled. The output of the scheduler is a microcode program of the DSP algorithm.

6.2.4 Register binding

Registers are used to hold the values of variables in the DSP algorithm. The lifetime of a variable is the interval of time from its birth to its death. The birth, or start cycle, of a variable is the cycle in which the value is created and the death, or stop cycle, of a variable is the last cycle in which it is used. A variable with a cyclic lifetime is defined as a variable whose start cycle is later than its stop cycle. A delay variable in a DSP algorithm is an example of a variable with a cyclic lifetime. Register binding is the process of assigning each variable to a register. Register binding minimizes the number of registers by allowing variables with non-overlapping lifetimes to share registers.

In Wizard, register binding takes place after scheduling is complete. The cost of assigning registers to a register file is weighed during the scheduling process. Therefore,
the only goal of the register binding algorithm is to compute the optimum register sharing. The left-edge algorithm is known to solve the register sharing problem in polynomial time, for variables with non-cyclic lifetimes. This research has developed a modified version of the left-edge algorithm which solves the register sharing problem for variables, including those with cyclic lifetimes. The register sharing problem with cyclic lifetime variables has been shown to be intractable [11], but this algorithm provides excellent results in polynomial time.

The modified left-edge binding algorithm is shown in Figure 6.5. In this algorithm, the variable structure contains three records: start, stop, and address, where start and stop refer to the start and stop time of the variables’ lifetime, and address refers to the address of the register used to store the variable. The first step of this algorithm is to separate variable with cyclic and acyclic lifetimes into two different sets. The sort function sorts the variables in each set. The cyclic variables are sorted in ascending order of their stop time as the first key, and in ascending order of their start times as the second key. The acyclic variables are sorted in ascending order of their start time as the first key, and in descending order of their stop times as the second key. The value of address is used to keep track of the current register being allocated. The values of begin and end are used to represent the available time interval of the current register.
$V_{cyclic} =$ set of variables with cyclic lifetimes;  
$V_{acyclic} =$ set of variables without cyclic lifetimes;  

SORT($V_{cyclic}$);  -- ascend stop, ascend start  
SORT($V_{acyclic}$);  -- ascend start, descend stop  

address = -1;  
while (|$V_{cyclic}$| > 0 AND |$V_{acyclic}$| > 0) {  
address = address + 1;  
begin = -1;  
end = $\infty$;  

if (|$V_{cyclic}$| > 0) {  
    var = first($V_{cyclic}$);  
    var.address = address;  
    begin = var.stop;  
    end = var.start;  
    $V_{cyclic} =$ $V_{cyclic} -$ var;  
}  

for all var $\in V_{acyclic}$ {  
if(var.start > begin AND var.stop < end) {  
    var.address = address;  
    begin = var.stop;  
    $V_{acyclic} =$ $V_{acyclic} -$ var;  
}  
}  

Figure 6.5: Modified left-edge algorithm for register binding

During each pass of the while loop, variables are assigned to a new register, identified by the value of address. If there is a variable with a cyclic lifetime, it is assigned to the register, begin and end are adjusted to reflect the available time interval of the
register, and the variable is removed from the set. Next, the variables with acyclic lifetime
are scanned in the sorted order. If the lifetime of a variable is contained by the available
time interval on the current register, it is assigned to the register, the begin cycle of the
available time interval is adjusted, and the variable is removed from the set. When the
algorithm terminates, all variables will be assigned to a register. This algorithm performs
the left-edge algorithm in the absence of cyclic variables.

An example of this algorithm is shown in Figure 6.6. This figure shows the sorted
lifetime intervals for cycle and acyclic variables. The algorithm starts off by assigning \( v1 \) to
register 0. The algorithm then travel through the sorted list of acyclic variables, adding
variables to register 0. When no more variables will fit into register 0, the algorithm starts
to assign variables to register 1. This process continues until all variables are assigned to a
register. The total set of variables is assigned to share five registers.
### Variables

<table>
<thead>
<tr>
<th>v1</th>
<th>v2</th>
<th>v3</th>
<th>v4</th>
<th>v5</th>
<th>v6</th>
<th>v7</th>
<th>v8</th>
<th>v9</th>
<th>v10</th>
<th>v11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.6: Register binding using the modified left-edge algorithm

### 6.3 Synthesis target architectures

The generic algorithms described in the previous section are used to synthesis for all FPDA architectures. The algorithms are customized for each FPDA target architecture, by changing the routine that determines when resources are available. The FPDA target architectures supported by Wizard include: custom, single-processor, and multi-processor.
The custom FPDA synthesis engine uses a time-constrained scheduler to minimize the hardware resource requirements. The custom scheduling algorithm does not put any constraints on the amount of hardware that can be allocated. Therefore, the custom synthesis engine is useful for performing application profiling.

The single-processor FPDA synthesis engine uses a resource-constrained scheduler to minimize the amount of time required to execute a DSP algorithm. The single-processor scheduler insures that the schedule meets resource constraints on the number of processing elements, internal buses, and registers. The single-processor scheduler differs from the multi-processor scheduler because it does not consider the effects of multi-processor communication.

There are two multi-processor FPDA synthesis engines, which use both scheduling algorithms: resource-constrained and time-constrained. The resource-constrained scheduler must meet the resource requirements of each processor, as described for the single-processor synthesis engine. The resource-constrained scheduler is also constrained by the number of processors available. The time-constrained multi-processor scheduler minimizes the number of FPDA processors needed to execute the DSP algorithm in a specified amount of time. Therefore, the time-constrained scheduler can allocate new processors as they are required.

Both multi-processor schedulers enforced a constraint on the number of function outputs per cycle. The number of function outputs per cycle is forced to be less than or equal to the number of input ports on the FPDA processor. As described in Section 5.3,
this requirement reduces the amount multi-processor communication and provides a natural partitioning of the DSP system.

6.4 Conclusions

The behavioral synthesis algorithms used by the Wizard behavioral synthesis system were described. These algorithms include: time-constrained synthesis, resource-constrained synthesis, and a modified left-edge algorithm for register binding. The application of these algorithms to the FPDA target architectures was also described.
The Wizard behavioral synthesis system provides a design environment for the FPDA architecture. Wizard supports FPDA processor development, through the custom processor synthesis engine. This synthesis engine can be used to perform application profiling, which matches resource requirements to the target application constraints. Wizard also supports DSP function development, through the single-processor synthesis engine. Finally, Wizard supports behavioral synthesis of DSP algorithms for single-processor and multi-processor FPDA systems.

In the first section of this chapter, the process of application profiling is demonstrated. In the second section, the single-processor synthesis engine is used to create a biquad filter function and this function is used to synthesize a system of parallel filters. In the third section, the multi-processor synthesis engine is used to implement the parallel filters on a multi-processor system.
7.1 Application profiling

When designing a new FPDA processor, application profiling is used to determine the processor resource requirements. DSP algorithms from the target application are synthesized under time constraints, in order to quantify the resource requirements. Wizard uses a time-constrained synthesis engine to determine the number and type of processing elements, the number of internal buses, and the number of registers. With the resource requirements set, a new FPDA processor can be designed. After creating a new entry in the target library, Wizard is prepared to generate microcode for the new processor. In this section, the development of two FPDA processors is described.

The DSP algorithms used for application profiling is depends on the target application. In these example, the algorithms used for application profiling are an 8th order symmetric FIR filter and a biquad filter. The VDFL description for these filters are shown in Figure 7.1 and Figure 7.2, respectively.
ENTITY fir IS
-- from Harris Semiconductor [37]

GENERIC(
  Width    : INTEGER;
  Fraction : INTEGER;
  Coeffs   : REAL();
);

PORT(
  IN1 : IN FIX<width, fraction>;
  OUT1 : OUT FIX<width, fraction>
);
END fir;

ARCHITECTURE vdfl OF fir IS
  SUBTYPE Data IS FIX<width, fraction>(0 TO 3);

  CONSTANT a : Data := Coeffs;
  SIGNAL prod : Data;

BEGIN

  prod(0) <= (IN1 + IN1@7) * a(0);
  prod(1) <= (IN1@1 + IN1@6) * a(1);
  prod(2) <= (IN1@2 + IN1@5) * a(2);
  prod(3) <= (IN1@3 + IN1@4) * a(3);

  OUT1 <= prod(0) + prod(1) + prod(2) + prod(3);

END vdfl;

Figure 7.1: VDFL description of an 8th order symmetric FIR filter
ENTITY biquad IS
   GENERIC(
      Width : INTEGER := 16;
      Fraction : INTEGER := 13;
      Coeffs : REAL();
   );
   PORT(
      IN0 : IN fix<Width, Fraction>;
      OUT0 : OUT fix<Width, Fraction>
   );
END biquad;

ARCHITECTURE vdf1 OF biquad IS
   SUBTYPE Data IS fix<Width, Fraction>;
   SUBTYPE Coef IS fix<Width, Fraction>(0 TO 3);

   CONSTANT a : Coef := Coeffs;
   SIGNAL state : Data;
BEGIN

   state <= IN0 + coef(0)*state@1 + coef(1)*state@2;
   OUT0 <= coef(3)*state + coef(2)*state@1;

END vdf1;

Figure 7.2: VDFL description of a biquad filter
7.1.1 Low-frequency FPDA processor

A low-frequency FPDA processor could target applications with low-power requirements and a low sample frequency. This processor could also be used as a DSP core to be included in a mixed-signal ASIC. The execution unit used in this processor is an add/shift unit. This unit provides addition, subtraction, and constant multiplication operations. Using the add/shift unit to perform multiplication reduces the throughput of the processor, but it also reduces the number of transistors required for a multiplier, thereby reducing the power requirements. The number of cycles required to execute an operation is as follows: adds and subtracts take one cycle, and multiplies take eight cycles for a 16x16 bit constant multiply.

Behavioral synthesis results for the FIR filter are shown in Table 7.1 and results for the biquad filter are shown in Table 7.2. The CPU time, shown throughout this chapter, indicates the amount of time required to run the synthesis program on a Pentium-133 workstation under Windows 95. The CPU time includes time to: read the SFG file, read the execution unit library, read the function library, read the target library, perform module selection, perform time-constrained scheduling and binding, and write the results to a file.

The first entry in each table indicates the resources required to match the critical path of the algorithm. The last entry in each table indicates the time required to schedule the algorithm using one instance of each processing element and one internal bus. All other entries indicate time constraints where the resource requirements change. For
example, in Table 7.1, all time constraints between 29 and 45 will have the same resource requirements as a time constraint of 29.

The resource constraints for a new FPDA processor are chosen based on the desired frequency response of the synthesized algorithms. The frequency response depends on the clock frequency of the FPDA processor. The frequency response of an algorithm is given by the formula:

\[
freq\_response = \frac{clock\_freq}{2*cycles}
\]

For this example, the low-frequency FPDA processor is given a system clock frequency of 10 MHz. The maximum frequency response for the FIR and biquad filters is shown in Figure 7.3 and Figure 7.4. Assuming a desired frequency response of 200 kHz for both filters, the number of add/shift units will be set to two and the number of internal buses will be set to two.
### Table 7.1: Resource requirements for FIR filter on low-frequency FPDA processor

<table>
<thead>
<tr>
<th>Time Constraint</th>
<th>Resource Requirements</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add/Shift</td>
<td>Bus</td>
</tr>
<tr>
<td>13</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>18</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>22</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>29</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>46</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Table 7.2: Resource requirements for biquad filter on low-frequency FPDA processor

<table>
<thead>
<tr>
<th>Time Constraint</th>
<th>Resource Requirements</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add/Shift</td>
<td>Bus</td>
</tr>
<tr>
<td>19</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>37</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

93
Figure 7.3: Maximum frequency response of FIR filter on low-frequency FPDA

Figure 7.4: Maximum frequency response of biquad filter on low-frequency FPDA
7.1.2 High-frequency FPDA processor

A high-frequency FPDA processor could target applications with high sample rates. The execution units used in this processor will include a add/subtract unit and a pipelined multiplier. The execution unit records for the add/sub and the multiplier units are shown in FIGURE. These records indicate that adds and subtractions take one cycle, while multiplies take two cycles, but a new multiply can be started on every cycle. Behavioral synthesis results for the FIR filter are shown in Table 7.3 and results for the biquad filter are shown in Table 7.4.

For this example, the high-frequency FPDA processor is given a system clock frequency of 100 MHz. The maximum frequency response of the FIR and biquad filters is shown in Figure 7.5 and Figure 7.6, respectively. Assuming a requirement of maximum throughput for both filters, the high-frequency FPDA resources are: two multipliers, two adders, and two internal buses.
<table>
<thead>
<tr>
<th>Time Constraint</th>
<th>Resource Requirements</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mult</td>
<td>Add/Sub</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 7.3: Resource requirements for FIR filter on high-frequency FPDA processor

<table>
<thead>
<tr>
<th>Time Constraint</th>
<th>Resource Requirements</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mult</td>
<td>Add/Sub</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 7.4: Resource requirements for biquad filter on high-frequency FPDA processor
Figure 7.5: Maximum frequency response of FIR filter on high-frequency FPDA

Figure 7.6: Maximum frequency response of biquad filter on high-frequency FPDA
7.2 Single-processor synthesis

Single-processor synthesis uses a resource-constrained synthesis engine to minimize the time required to execute a DSP algorithm on a single FPDA processor. This synthesis engine can also be used to generate DSP functions. DSP functions represent common, reusable DSP algorithms for the target application area. Functions are also used by the multi-processor synthesis engine to create a natural partitioning of a DSP system. In this section, a function will be created for the biquad filter. This function will be used to synthesize the system of parallel filters shown in Figure 7.8. A comparison between synthesis using functions and synthesis using a flattened SFG is also presented.

A function can be synthesized to utilize all or a portion of a processor’s resources. To synthesize a function using a portion of a processor’s resources, a new target entry is created, which indicates the subset of processor resources to be utilized. Table 7.5 shows the results of synthesizing the biquad filter for two resource subsets the low-frequency FPDA processor. The schedule results from Wizard are shown in Figure 7.7. Table 7.6 shows the results of synthesizing the biquad filter for two resource subsets the high-frequency FPDA processor.

Table 7.7 shows a comparison of synthesizing the system of parallel biquad filters. The table shows the number of cycles required to execute the parallel system described using biquad_1 filters, biquad_2 filters, and a flattened SFG. On both processors, the flattened description yields better synthesis results than either description using functions. This is a result of free processor cycles being wasted inside the function descriptions.
ENTITY biquad IS
GENERIC(
    Width : INTEGER := 16;
    Fraction : INTEGER := 15;
    Coefs : REAL);
END biquad;

ARCHITECTURE vdf1 OF biquad IS
SUBTYPE Data IS fix<Width, Fraction>;
SUBTYPE Coef IS fix<Width, Fraction>(0 TO 3);
CONSTANT a Coef = Coefs;
SIGNAL state Data;
BEGIN
    OUTO <= a(3) * state + a(2) * state1.
END vdf1.

Figure 7.7: Wizard schedule display for biquad filter
### Table 7.5: Biquad resource usage on the low-frequency FPDA

<table>
<thead>
<tr>
<th></th>
<th>Add/Shift</th>
<th>Buses</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>biquad 1</td>
<td>1</td>
<td>1</td>
<td>37</td>
</tr>
<tr>
<td>biquad 2</td>
<td>2</td>
<td>2</td>
<td>19</td>
</tr>
</tbody>
</table>

### Table 7.6: Biquad resource usage on the high-frequency FPDA

<table>
<thead>
<tr>
<th></th>
<th>Mult</th>
<th>Add</th>
<th>Buses</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>biquad 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>biquad 2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

### Table 7.7: Cycles required for the parallel biquad system using one FDPA

<table>
<thead>
<tr>
<th>Processor</th>
<th>biquad 1</th>
<th>biquad 2</th>
<th>flat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low-Freq.</td>
<td>77</td>
<td>79</td>
<td>76</td>
</tr>
<tr>
<td>High-Freq.</td>
<td>23</td>
<td>31</td>
<td>21</td>
</tr>
</tbody>
</table>

100
The choice of using a function which utilizes all of the processor’s resources versus a function that uses a subset of the processor’s resources depends on the DSP system. For this example, biquad_1 is preferred over biquad_2 because the system contains parallel biquad filters. The resource subset used by the biquad_1 function allows two filters to be executed in parallel on one processor. For systems of a sequential nature, a function that uses all resources of the FPDA processor would provide better results, by reducing the amount of time for each function in the sequence.

7.3 Multi-processor synthesis

Wizard provides two multi-processor synthesis engines. One engine uses a resource-constrained synthesis algorithm to minimize the time required to execute a DSP algorithm. The other engine uses a time-constrained synthesis algorithm to determine the number of FPDA processors needed to execute a DSP algorithm. Functions are used by the multi-processor synthesis engine to create a natural partitioning of a DSP system. In this section, the functions created in the previous section will be used to demonstrate the multi-processor synthesis engines.

In order to support the multi-processor architecture, the FPDA processors must be modified. The low-frequency processor will maintain one input port, and that port will be connected to both processing elements. The high-frequency processor will increase to two input ports, which are connected to all processing elements.
In this example, the system of parallel biquad filters, shown in Figure 7.8 is synthesized. The results of resource-constrained synthesis for both processors are shown in Table 7.8. These results show a speedup of 1.8 is possible when using two processors. The greatest speedup is obtained when using four processors and the biquad_2 description. This combination maps each biquad filter to a separate processor, therefore the faster sequential execution time of the biquad_2 function yields a faster execution time.

The time-constrained synthesis engine provides the same synthesis results as the resource-constrained synthesis. For example, supplying a time-constraint of 50 cycles for
the low-frequency FPDA processor indicates that two processors are required. Supplying
a time-constraint of 15 cycles for the high-frequency FPDA processor indicates two
processors for the biquad_1 description and four processors for the biquad_2 description.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Function</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Low-Freq.</td>
<td>biquad_1</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td>biquad_2</td>
<td>79</td>
</tr>
<tr>
<td>High-Freq.</td>
<td>biquad_1</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>biquad_2</td>
<td>31</td>
</tr>
</tbody>
</table>

Table 7.8: Cycles required for the parallel biquad system using multiple FPDA's

7.4 Conclusions

Design examples using the Wizard behavioral synthesis system have been
discussed. Application profiling was used to develop new FPDA processors for low-
frequency and high-frequency target applications. A biquad filter function was developed
for each processor, and a system of parallel biquad filters was synthesized on a single-
processor and multi-processor implementations.
CHAPTER 8

CONCLUSIONS

A microcode programmable, application specific DSP architecture and the behavioral synthesis system designed to support the use of this architecture have been presented. The Field Programmable DSP Array (FPDA) architecture has been developed to provide a flexible and extendible target architecture for behavioral synthesis. The FPDA architecture allows the addition of application specific execution units, so an FPDA processor can be designed to meet the requirements of the target application. A multi-processor extension to the FPDA architecture has been described, as well as an efficient method to synthesize multi-processor systems. The Wizard behavioral synthesis system has been developed to support FPDA design activities. New synthesis algorithms for the FPDA target have been developed and implemented in the Wizard synthesis tool.
Specific contributions of this research to the behavioral synthesis of real-time DSP processors include:

1. The application specific Field Programmable DSP Array (FPDA) processor target architecture. The FPDA architecture supports single-processor and multi-processor implementations.

2. The VHDL Data Flow Language (VDFL) for behavioral specification of DSP algorithms.

3. Efficient list-based synthesis algorithms that perform simultaneous scheduling and binding for the FPDA target architecture.

4. The modified left-edge algorithm for solving the register sharing problem on variables with acyclic lifetimes.

5. The Wizard behavioral synthesis tool, which can be used to design new FPDA processors, develop DSP functions, and generate microcode for FPDA processors.

The Wizard synthesis system provides a flexible design environment for developing FPDA processors and systems. Possible future research for this project includes:

1. Development of FPDA processors for specific target applications.

2. Population of the function library for the new FPDA processors.


4. Development of evaluation board for real-time prototyping of DSP systems.
BIBLIOGRAPHY


