ARCHITECTURAL OPTIMIZATION FOR FLUID IP CORE GENERATORS

Jeffrey Alan Geib

Coordinated Science Laboratory
1308 West Main Street, Urbana, IL 61801
University of Illinois at Urbana-Champaign
ARCHITECTURAL OPTIMIZATION
FOR FLUID IP CORE GENERATORS

Jeffrey Alan Geib

Coordinated Science Laboratory
1308 West Main Street, Urbana, IL 61801
University of Illinois at Urbana-Champaign
# REPORT DOCUMENTATION PAGE

**Title:** Architectural Optimization for Fluid IP Core Generators

**Author:** Geib, Jeffrey Alan

**Performing Organization:**
Coordinated Science Laboratory  
University of Illinois at Urbana-Champaign  
1308 W Main St  
Urbana, IL 61801

**Sponsoring/Monitoring Agency:**
NSF/CISE/C-CR Rm 1145  
4201 Wilson Blvd  
Arlington, VA 22230

**Abstract:**
Architecture optimization and pipeline stage optimization techniques are presented within a fluid IP core generator framework. From inputs of power and delay models of a fluid cell library, a library of template transformations, algorithmic specifications, and power and delay specifications, a custom-quality layout of an algorithm-specific block is generated without going through a synthesis and place-and-route step. To meet throughput requirements and reduce power consumption, the architecture optimizer applies transform templates to a data-flow graph (DFG) of a desired filter type. The pipeline stage optimizer then reduces the transistor sizes in each pipeline stage to minimize power consumption. This methodology is shown to provide significant performance benefits over a traditional synthesis design flow, while allowing migration to future technologies.

**Subject Terms:**
Algorithm transforms, core generator, transistor sizing, voltage scaling, ASIC design, FIR filter

**Security Classification:**
- **UNCLASSIFIED**

**Funding Numbers:**
- NSF CCR 00-00987

**Performing Organization Report Number:**
- UULU-ENG-03 (DAC 98 ) 2206
- NSF CCR 00-00987

**Distribution/Availability Statement:**
Approved for public release; distribution unlimited.

**Number of Pages:**
74
ARCHITECTURAL OPTIMIZATION
FOR FLUID IP CORE GENERATORS

Jeffrey Alan Geib

Coordinated Science Laboratory
1308 West Main Street, Urbana, IL 61801
University of Illinois at Urbana-Champaign
ARCHITECTURAL OPTIMIZATION
FOR FLUID IP CORE GENERATORS

BY

JEFFREY ALAN GEIB
B.S., University of Toledo, 2000
B.S., University of Toledo, 2000

THESIS
Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2002

Urbana, Illinois
ACKNOWLEDGMENTS

This thesis would not have been possible without the help of Tim Wilson and Eric Martina, my fellow team members on this project. I would also like to express my gratitude to my advisor, Naresh Shanbhag, for his help and guidance.

Finally, I must thank my parents, Brian and Kathy, for their love and support.
# TABLE OF CONTENTS

1 INTRODUCTION ........................................... 1  
  1.1 Motivation ........................................... 1  
  1.2 Fluid IP Core Generator Components ................. 2  
    1.2.1 Fluid core optimizer ............................ 2  
    1.2.2 Power and delay models ......................... 4  
    1.2.3 Layout synthesis .................................. 5  
  1.3 Outline ............................................... 6  

2 ARCHITECTURE OPTIMIZER ............................... 7  
  2.1 Architecture Generation ............................. 9  
    2.1.1 Representation .................................... 9  
    2.1.2 Implementation ................................... 11  
    2.1.3 Critical path calculation ...................... 12  
  2.2 Template Transforms .................................. 13  
    2.2.1 Architectural strength reduction ............... 13  
    2.2.2 Pipelining ....................................... 17  
    2.2.3 DECOR ........................................... 19  
  2.3 Voltage Scaling ...................................... 23  
  2.4 Optimizer Output ..................................... 24  

3 PIPELINE STAGE OPTIMIZATION VIA DELAY BALANCING .... 26  
  3.1 Fluid Full Adder Cell ................................ 28  
    3.1.1 Architecture ...................................... 28  
    3.1.2 Delay model ...................................... 29  
  3.2 Delay Balancing ...................................... 32  
    3.2.1 Ripple-carry adder delay balancing ............... 32  
    3.2.2 Baugh-Wooley multiplier delay balancing ........ 33  
    3.2.3 Unpipelined multiplier-adder delay balancing ... 35  
    3.2.4 Pipelined multiplier-adder delay balancing ...... 38  
  3.3 Results .............................................. 41  

iv
4 TEST CHIP ................................................................. 45
  4.1 Filter Descriptions ................................................. 45
    4.1.1 Fluid IP core generator filters ............................... 47
    4.1.2 Synthesis benchmark ............................................ 49
  4.2 Chip Architecture .................................................. 49
  4.3 Results .................................................................. 50
  4.4 Summary ................................................................. 52

APPENDIX A  CORE OPTIMIZER SOURCE CODE OUTLINE ....... 54

APPENDIX B  VHDL SOURCE CODE FOR TEST CHIP .......... 58

REFERENCES ................................................................. 64

VITA ........................................................................... 66
LIST OF TABLES

2.1: DFG node properties. ................................................. 10
2.2: DFG edge properties. ................................................. 10
2.3: $\alpha$ and $\beta$ for different types of FIR filters. ................. 20

3.1: Adder delay balancing results. ....................................... 43
3.2: Multiplier delay balancing results. ................................... 43
3.3: Unpipelined tap delay balancing results. ............................ 44
3.4: Fine-grain pipelined tap delay balancing results. .................... 44
3.5: Synthesis multiplier power comparison. .............................. 44

4.1: Simulation results from test chip filters. .............................. 52
# LIST OF FIGURES

1.1: Fluid IP core generator structure. ........................................... 3

2.1: Architecture optimizer structure. ........................................... 9
2.2: Example of adjacency list representation: (a) DFG, (b) corresponding adjacency list representation. ........................................... 11
2.3: Multiply-accumulate structure and critical path. ......................... 13
2.4: Template transformations in the architecture optimizer. .................. 14
2.5: Architectural strength reduction: (a) original complex FIR filter architecture, (b) strength-reduced architecture. ........................................... 16
2.6: Cutsets for pipelining to the tap level. ...................................... 18
2.7: Cutsets for pipelining after the multipliers. .................................... 18
2.8: Cutset for fine-grained pipelining of multiplier-adder structures. .......... 19
2.9: Decorrelating (DECOR) transform: (a) strength-reduced complex FIR filter architecture, (b) after application of DECOR. .................. 20
2.10: Precision reduction after applying DECOR to a 40-tap, 16-bit real low-pass filter and complex low-pass filter (centered at $\omega = \pi/10$). ........................................... 22
2.11: Precision reduction after applying DECOR to a 40-tap, 16-bit complex high-pass filter (centered at $\omega = 9\pi/10$). ........................................... 22
2.12: Precision reduction after applying DECOR to a 40-tap, 16-bit complex band-pass filter (centered at $\omega = \pi/5$). ........................................... 23
2.13: Conversion of DFG to HDFG: (a) data-flow graph, (b) equivalent hardware data-flow graph (HDFG). ........................................... 25

3.1: Fluid full adder cell schematic. ........................................... 28
3.2: Fluid full adder cell layout, with groups of scaling transistors labeled. ........................................... 30
3.3: Split fluid full adder cell layout for delay model creation. ................... 31
3.4: Five-bit adder with critical path highlighted. ................................... 34
3.5: Five-bit adder with full adder sizes after scaling. ................................... 35
3.6: $8 \times 12$ Baugh-Wooley multiplier, with critical path highlighted and scaling regions indicated. ........................................... 36
3.7: Referencing of cells in Baugh-Wooley multiplier. ................................... 38
3.8: Algorithm for sizing cells in a Baugh-Wooley multiplier. .................... 39
3.9: Baugh-Wooley multiplier scaling results. ........................................... 40
3.10: Layout of Baugh-Wooley multiplier: (a) before scaling, (b) after scaling. ........ 40
3.11: Multiplier-adder structure, with critical path highlighted. \hspace{1cm} 41
3.12: Pipelined multiplier-adder structure, with pipelining cutset and critical path highlighted. \hspace{1cm} 42

4.1: Frequency and phase response of test chip filters. \hspace{1cm} 46
4.2: Clock routing strategy. \hspace{1cm} 48
4.3: Architecture of fluid IP core generator test chip. \hspace{1cm} 50
4.4: Layout of fluid IP core generator test chip. \hspace{1cm} 51
CHAPTER 1
INTRODUCTION

1.1 Motivation

When designing DSP systems, several conflicting goals are encountered. An ideal process flow produces high-throughput, low-power, area-efficient designs in a short design cycle. Current design methodologies can be broadly classified as logic synthesis-based, custom, and the more recent core-based methodology. Full-custom methodologies produce the highest-performance designs available, allowing the designer full control over the algorithms, architecture, and transistor sizes. However, this flexibility comes at the price of long design cycles. Conversely, synthesis-based methodologies have very short design cycles by incorporating a logic synthesis step, followed by automatic placement and routing of standard cells. Hardware description language (HDL) inputs provide a high level of flexibility, as designs can be quickly modified and resynthesized. However, synthesis designs suffer from unpredictable quality, sacrificing throughput, power, and area for this short design cycle. As designs increase in complexity and reach deep submicron levels, noise and interconnect delay have an increasingly detrimental effect.

Core-based designs attempt to preserve the short design cycle of synthesis-based designs while providing predictable and efficient designs. These designs consist of predesigned and verified cores, which are placed together to form a system-on-a-chip. Soft cores are flexible and portable, but suffer from many of the same problems as synthesis designs. Hard cores sacrifice flexibility and portability to provide designs of predictable quality.
A fluid IP core generator builds upon the idea of core-based design methodology by synthesizing hard cores with application-specific properties. A fluid IP core generator synthesizes a custom-quality layout of an algorithm-specific block from high-level specifications without going through a synthesis and place-and-route step. The term \textit{fluid} refers to the fact that the core generator, while optimizing the architecture, can reach down into the circuit fabric and tune transistor sizes while generating layouts. The core generator brings optimizations from the algorithm, architecture, and circuit domains and makes them available to the average designer.

\section*{1.2 Fluid IP Core Generator Components}

A fluid IP core generator uses a core-based design methodology to synthesize hard cores with application-specific properties. Figure 1.1 illustrates the architecture of the fluid IP core generator. The fluid IP core generator uses a fluid core optimizer to generate filter architectures based on user inputs of algorithmic and power/delay specifications. The core generator has two major components: an architecture optimizer and a layout synthesizer. The core generator accepts as inputs power and delay models, a library of template transforms, algorithmic specifications, and power and delay specifications.

\subsection*{1.2.1 Fluid core optimizer}

The fluid core optimizer applies algorithm transformation techniques to optimize the throughput and power. In doing so, the optimizer is also aware that it can tune the transistor sizes. The fluid core optimizer consists of an architectural optimizer and a pipeline stage optimizer.
Architectural optimization via transforms

The architecture optimizer is discussed in depth in Chapter 2. The fluid core optimizer begins the design process with a data-flow graph (DFG) based on the desired filter type. The DFG represents computational units (such as multipliers and adders) as nodes and the flow of data between the computational units as edges. Architecture-specific information, such as word sizes and delays, is embedded in the graph. Power and delay models are used to evaluate architectures and compare to user specifications.

Transform templates, such as strength reduction, pipelining, and DECOR, are applied to meet the throughput requirements and to reduce power consumption. Also, throughput requirements can be exceeded, and voltage scaling can be used to reduce power consumption [1]. Architectural transformations are chosen that preserve the regularity of the filter structures. By doing so, the layout synthesis techniques described below can generate regular layouts with limited interconnect.
Pipeline stage optimization via delay balancing

Pipeline stage optimization is discussed in depth in Chapter 3. After the application of transform templates, the fluid core optimizer reduces transistor sizes in the fluid cells of each pipeline stage to minimize power consumption. The fluid cell library consists of one parameterized cell for every logic cell in the library, and each cell can be instantiated at virtually any size desired. This enhances the ability to optimize for delay and power. The fluid cell library also has cells with multiple scale factors, each corresponding to a different group of transistors within the cell. Thus in a full adder cell, the delay through the carry and sum paths can be individually optimized. The core generator uses ripple-carry adders and Baugh-Wooley multipliers due to their regularity in layout. These adder and multiplier architectures have many paths shorter than the critical path. The shorter paths can be exploited to reduce power by employing the scaling factors to shrink the transistors in cells off the critical path. The core optimizer contains algorithms to size cells in the multipliers, adders, and MACs.

1.2.2 Power and delay models

It is infeasible to characterize each cell for every possible instantiated size. In [2], models that describe the delay and power of the cells in terms of the scaling parameters of each cell are developed for the fluid core generator, using the techniques described in this section.

The macromodeling [3],[4] approach is used to model the delay of cells, modified to include transistor sizing. This results in equations describing the delay from each input to each output of a cell, in terms of the size of the cell and the capacitive load, and equations describing the input capacitance of each cell input. Architectural delay models are built from the cell-level models and incorporate interconnect delay.

Power models for the core generator are built from the Dual Bit Type [5] method, modified to include cell scaling. This approach is based on the observation that high-order bits in a
word act as sign bits, exhibiting a relation to word-level temporal correlation, while the lower-order bits behave randomly, cycling quickly for small changes in the word. Two breakpoints are defined to describe this behavior, and linear extrapolation is employed to model the bits between the two breakpoints. Word-level statistics such as mean, standard deviation, and temporal correlation are used to define the breakpoints. At the architectural level, statistics propagation [6] is used to determine the word-level statistics at each note.

1.2.3 Layout synthesis

The layout synthesizer accepts architectural parameters and transistor scaling factors from the core optimizer to synthesize a layout, as described in detail in [7]. As the architectural template is known, the synthesizer starts off with a good initial placement and routing. It then sizes the power grid and designs the clock tree.

From the power estimates of the filter and taps, the synthesizer calculates the number of power and ground lines and their dimensions.

Priority is given to short, regular interconnect, and then to area. Short interconnect is important because a design focuses is on low power consumption, and as interconnect capacitance goes down, power does as well. The regularity of the interconnect is necessary so that the models can accurately predict the output capacitance of a full adder cell and then use that to determine the power and delay of the cell.

To reduce interconnect length, it is necessary to keep the cells as close together as possible while still keeping a regular structure. Therefore, even though interconnect was given a higher priority, the overall area is still reduced when compared with synthesis.

At the end, the layout synthesis routes and buffers the clock. The size of each buffer is determined by the amount of load that the buffer is expected to face. The synthesizer forces the direction of clock routing to be in the opposite direction to that of the data in order to combat the effect of skew.
1.3 Outline

This work describes the fluid core optimizer of a fluid IP core generator. Chapter 2 discusses the architecture optimizer, and Chapter 3 discusses the pipeline stage optimizer. Finally, Chapter 4 describes a test chip that demonstrates the benefits of a fluid IP core generator and the combined benefits of architecture and pipeline stage optimization.
CHAPTER 2

ARCHITECTURE OPTIMIZER

Algorithm transformations improve the throughput, reduce the power consumption, and reduce the area of the VLSI implementation of a given algorithm [8],[9]. A large number of algorithm transformations exist, including architectural strength reduction, pipelining/retiming, DECOR, folding, unfolding, and block processing [10],[11],[12]. These techniques jointly optimize algorithm and VLSI performance by modifying the algorithm structure to introduce VLSI-friendly features.

Traditionally, algorithm transforms are applied by hand. While this allows a designer to improve an algorithm’s performance, it requires knowledge of the transformation as applied to the particular circuit, as well as the design time required to apply and validate the resulting algorithm. While certain transforms can be described within an HDL representation for synthesis, this again involves tailoring the transform for the particular design, along with verification.

By combining architectural transformation techniques with circuit-level optimizations of Chapter 3 in the fluid IP core generator methodology, the average designer quickly and easily reaps their benefits. Since the core generator targets specific blocks, transform template modules can be developed that encapsulate the details of each transform’s application to those blocks. Within this automated framework, the designer is able to explore performance points throughout the design space.

Previous works attempt to automate portions of the design process. FIRGEN [13] is a filter generator that incorporates CAD tools to automate the filter design process. However,
this method relies on a predefined set of arithmetic units and thus cannot approach the quality of custom designs. The MMAlpha tool automates the architecture design of LMS adaptive filters [14]. It does not optimize for power or area, though, and is targeted for ASICs and FPGAs. It will therefore suffer from the throughput and power penalties of these approaches. MetaCores [15] provides optimization techniques for algorithm selection in Viterbi decoders and IIR filters, but does not specifically optimize for power and does not avoid synthesis.

In the fluid IP core generator, the architecture optimizer guides the application of algorithm transforms to specific architecture blocks. Figure 2.1 shows the structure of the architecture optimizer. The user inputs are applied to generate an initial architecture, where a data-flow graph (DFG) holds the architectural representation. To achieve the performance of custom layouts, the architecture optimizer applies a series of algorithm transformations until the user specifications are met. The optimizer uses transform templates, which define the application of an algorithm transformation to a specific architecture. Power and delay models evaluate the architecture after transform template application. By drawing on a library of template transformations, the optimizer can repeatedly apply transforms until user specifications on throughput and power are met.

This chapter describes the implementation of the architecture optimizer for complex finite impulse response (FIR) filters. Section 2.1 describes the internal representation and filter architecture generation. Section 2.2 describes algorithm transforms and their implementation as template transforms. Finally, Section 2.4 discusses the output the architecture optimizer provides to the designer and the layout synthesis stage.
2.1 Architecture Generation

2.1.1 Representation

A DFG is used to represent the architecture information. The DFG contains the mapping of algorithm information to computational units in the layout, the flow of data between the computational nodes, and the delays between those nodes. A DFG is a directed graph $G = (V, E, w, v)$, where $V$ is the set of nodes representing computational units, $E$ is the set of edges representing the flow of data from one computational unit to another, $w$ is the set of edge weights (equal to the number of delays on that edge), and $v$ is the set of node delays (equal to the time to compute one sample of output).

In addition to the edge weights and the node delays, the architecture optimizer attaches other properties to both the nodes and the edges. These properties serve to uniquely define the architecture by adding information about data precisions, computational unit types, and cell sizes. Additionally, input/output labels are attached. These properties are summarized
Table 2.1: DFG node properties.

<table>
<thead>
<tr>
<th>Node property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Specifies node as input, output, adder, multiplier, or dummy</td>
</tr>
<tr>
<td>Label</td>
<td>Text label, specifying input, output, or coefficient name</td>
</tr>
<tr>
<td>Bits</td>
<td>Size of adder, or size of the first input of a multiplier</td>
</tr>
<tr>
<td>Bits2</td>
<td>Size of second input of multiplier</td>
</tr>
<tr>
<td>Delay</td>
<td>Node delay</td>
</tr>
<tr>
<td>Alpha</td>
<td>Size of full adder cells in adder or multiplier (before scaling)</td>
</tr>
<tr>
<td>AlphaCArray</td>
<td>Array of carry stage full adder cell sizes in adder or multiplier (after scaling)</td>
</tr>
<tr>
<td>AlphaSArray</td>
<td>Array of sum stage full adder cell sizes in adder or multiplier (after scaling)</td>
</tr>
</tbody>
</table>

Table 2.2: DFG edge properties.

<table>
<thead>
<tr>
<th>Edge property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delays</td>
<td>Edge delay</td>
</tr>
<tr>
<td>Bits</td>
<td>Precision of signal on the edge</td>
</tr>
<tr>
<td>Num</td>
<td>Unique edge number (for producing VHDL output)</td>
</tr>
</tbody>
</table>

in Tables 2.1 and 2.2 and described below.

The Type property specifies one of five types: input, output, adder, multiplier, or dummy. Input and output nodes represent the interface between the architecture and the external world. A dummy node is a node that does no computation, but simply broadcast signals. Adder and multiplier nodes represent computational units of the same type. The Bits property represents the size of the computation unit, either the precision of an adder or the first input precision of a multiplier. For multipliers, Bits2 represents the precision of the second input. The Delay property stores the actual delay through an adder or multiplier node, as computed by the delay models. The Alpha property is the initial scaling of the fluid full adder cells in an adder or multiplier. AlphaCArray and AlphaSArray store the sizes of the full adder cells in an adder or multiplier after the transistor sizing performed by the fluid cell optimizer, as described in Chapter 3.

Edges have three properties: Delays, Bits, and Num. Delays stores the number of latches on that edge, while Bits stores the precision of the data on that edge. Num contains a unique number for each edge, used to uniquely label the signals between the nodes when creating a VHDL description of the architecture.
Figure 2.2: Example of adjacency list representation: (a) DFG, (b) corresponding adjacency list representation.

The DFG is stored internally as an adjacency list. An adjacency list contains a list of vertices for each vertex. Vertex $i$ is contained in vertex $j$’s list if there exists an edge from vertex $j$ to vertex $i$. An example representation is shown in Figure 2.2. This structure is compact for sparse graphs, like those encountered in DSP architectures, and allows quick traversal of graphs by following the links between vertices.

2.1.2 Implementation

The architecture optimizer generates the initial architecture based on the input precision $P_i$, the coefficient precisions $P_c$, and number of taps $N$. The architecture is either a direct form or a transpose data-broadcast structure. The required accumulator precision $P_a$ to prevent overflow is calculated as

$$P_a = [P_i + P_c + \log_2(N)]$$ (2.1)

With this information, the precision of all signals and nodes is known. Nodes and edges, with the appropriate properties, are inserted to complete the graph of the architecture. Input nodes are given labels with the suffix .in, such as $xx$.in, and output nodes are given labels with the suffix .out, such as $yi$.out. Coefficients are represented by an input node with edges
to the appropriate multipliers. Real coefficients are labeled \(c_1\) through \(c_N\), and imaginary coefficients are labeled \(d_1\) through \(d_N\).

### 2.1.3 Critical path calculation

A path \(p\) of the DFG is a sequence of nodes and edges \(u \rightarrow v \rightarrow \cdots \rightarrow w\), where \(u\) is the source node and \(w\) is the destination node of the path. The iteration period (IP) of the DFG is given by

\[
IP = \max_{\forall p \in \text{aDFG}} d(p) \tag{2.2}
\]

where aDFG is an acyclic version of the original DFG, obtained by removing edges with nonzero delays, and \(d(p)\) is the sum of the node delays in \(p\). The critical path of a DFG is a path \(p\) such that \(d(p) = IP\).

To determine the critical path of the architecture, the delay of each adder and multiplier is calculated and stored in the Delay property of the node. The Alpha property is used as the size of all of the full adder cells in the filter. Later, the cells are scaled to balance the delay and lower the power consumption, as discussed in Chapter 3. However, cell sizes in the critical path are not modified during fluid cell scaling. Thus, a single cell size still provides accurate delays.

Where an adder directly follows a multiplier, the delay through the two is less than the sum of the individual delays. This is evident in Figure 2.3. When the highlighted critical path exits the multiplier, it passes through only the final few bits of the adder structure. When populating the delay of the nodes, if an unpipelined multiply-accumulate structure is encountered, the adder delay is adjusted to be only the delay through the bits following the multiplier output.

After the node delays have been populated, a copy of the graph is made. All edges with delays are removed, and all node delays are negated. An all-pairs shortest path algorithm is run on the graph. Since the delays were negated, the shortest path returned is the critical
Figure 2.3: Multiply-accumulate structure and critical path.

path of the architecture.

2.2 Template Transforms

The architectural optimizer uses a library of template transforms to modify the architecture of the filter. Each filter type has transform modules specifically tailored for it, as shown in Figure 2.4. While the transform templates are general enough to be applied to filter of different precisions and sizes, providing specific modules for each filter type simplifies their design. Additionally, the modular approach allows new transforms to be added to the library.

Pipelining, strength-reduction, and DECOR transforms are available for complex FIR filters. Additionally, pipelining can be combined with voltage scaling for low-speed, low-power designs. The background of each transform is discussed below, along with a description of its application by the transform template.

2.2.1 Architectural strength reduction

For complex filter structures, strength reduction reduces the number of multiplications and additions, reducing area and power consumption. Algorithmic strength reduction re-
Figure 2.4: Template transformations in the architecture optimizer.

duces the number of multiplications at the expense of an increase in addition operations. Since the size of a multiplier is much less than that of an adder, the net result is a reduction in area and power. As an example, the following complex multiplication example requires four multiplications and two additions in a direct implementation:

\[(a + jb)(c + jd) = ac - bd + j(ad + bc) = e + jf\]  \hspace{1cm} (2.3)

The following identities can be used:

\[ac - bd = a(c - d) + d(a - b)\]
\[ad + bc = b(c + d) + d(a - b)\]  \hspace{1cm} (2.4)

Thus, the original complex multiplication can be written as

\[a(c - d) + d(a - b) + j(b(c + d) + d(a - b)) = e + jf\]  \hspace{1cm} (2.5)
Here, the arithmetic complexity has been reduced to three multiplications and three additions, trading a multiplication for an addition.

Architectural strength reduction exploits the fact that filtering is equivalent to polynomial multiplication. Assume the filter input to be a complex signal $X(n)$ given by

$$X(n) = X_r(n) + jX_i(n) \quad (2.6)$$

where $X_r(n)$ and $X_i(n)$ are the real and imaginary parts, respectively. The filter coefficients are a complex vector $W$ given by

$$W = c + jd \quad (2.7)$$

where $c$ and $d$ are the real and imaginary parts respectively. The output $y(n)$ of a complex filtering operation can then be written as

$$y(n) = W^H X(n)$$
$$= [c^T X_r(n) + d^T X_i(n)] + j[c^T X_i(n) - d^T X_r(n)]$$
$$= y_r(n) + jy_i(n) \quad (2.8)$$

The filtering operation in (2.8) involves multiplying two complex polynomials. Thus, strength reduction as presented above can be applied

$$y(n) = [y_1(n) + y_3(n)] + j[y_2(n) + y_5(n)] \quad (2.9)$$

where

$$y_1(n) = c^T (X_r(n) - X_i(n))$$
$$y_2(n) = -d^T (X_r(n) + X_i(n))$$
$$y_3(n) = (c^T + d^T) X_i(n) \quad (2.10)$$
Figure 2.5: Architectural strength reduction: (a) original complex FIR filter architecture, (b) strength-reduced architecture.

This architecture is shown in Figure 2.5 and requires three filters and two output adders. This corresponds to $3N$ multipliers and $4N$ adders, where $N$ is the number of taps in the filter. The power savings is given by [10]

$$PS = \frac{P_o - P_{sr}}{P_o} = \frac{(2NK_C - 3)}{4(2NK_C + 3N)}$$

(2.11)

where $P_o$ is the dynamic power dissipation of the original architectures, $P_{sr}$ is the dynamic power dissipation of the strength-reduced architecture, and $K_C$ is the ratio of the capacitance of a multiplier to that of an adder [10]. As the number of taps increases, this approaches a power savings of 25%.

Since strength reduction so dramatically alters the filter architecture, the architecture optimizer presents it as a separate initial architecture. A module builds the nodes and edges of the strength-reduced architecture from the input and coefficient precisions and the number of filter taps. With this approach, strength reduction is applied before any other transform.
2.2.2 Pipelining

Pipelining is a technique to increase the throughput of an algorithm. By appropriately inserting delays on the edges of a DFG, the IP of a DFG can be reduced. The resulting DFG will have several pipelining stages that operate concurrently.

For nonrecursive DFGs, pipelining is straightforward. To do so, the feedforward cutset pipelining technique is used [16]. A cutset of a DFG is defined as a set of edges in a DFG that when removed results in two disjoint DFGs. A feedforward cutset is a cutset in which the source nodes of all the edges lie in one subgraph, and the destination nodes of all the edges lie in a second disjoint subgraph. To pipeline a DFG, feedforward cutsets are identified. Delays can then be added to all the edges in the feedforward cutset.

The architecture optimizer allows pipelining at three different levels: in between taps, in between the adders and multipliers of the taps, and two-stage fine-grained pipelining of the multiplier-adder structures. Figure 2.6 shows the cutsets required to pipeline in between taps. To pipeline the filter at this tap level, the architecture optimizer searches for the multiply-accumulate structure inherent to taps, and adds delays to the out-edges of the taps.

To pipeline in between the adders and multipliers, the DFG is first pipelined to the tap level. Next, delays are placed along the feedforward cutsets shown in Figure 2.7. Delays are added to the edge following each multiplier. The optimizer then finds the last tap of each filter section and adds delays to the edges exiting the accumulators.

For fine-grained pipelining of the multiplier-adder structures, tap-level pipelining is performed until two delays appear after each tap adder. The cutset in Figure 2.8 performs the fine-grained pipelining. Details of the cutset within the multiplier-adder structure are shown in Chapter 3.
Figure 2.6: Cutsets for pipelining to the tap level.

Figure 2.7: Cutsets for pipelining after the multipliers.
2.2.3 DECOR

Decorrelating (DECOR) transforms decorrelate the transfer function or inputs of a digital filter, such that fewer bits are required to represent the coefficients and inputs [11]. This reduces the sizes of arithmetic units and the number of signal transitions within, reducing power consumption.

For a fixed coefficient filter, the transfer function $H(z)$ is multiplied and divided by the polynomial

$$f(z) = (1 + \alpha z^{-\beta})^m$$

(2.12)

where $\alpha$ is the decorrelating coefficient, $\beta$ is the decorrelating period, and $m$ is the decorrelation order, all chosen depending on the frequency characteristics of the transfer function. Values of $\alpha$ and $\beta$ for different filter types are shown in Table 2.3. The $z$-transform of the filter output is given by

$$Y(z) = H(z) \frac{(1 + \alpha z^{-\beta})^m}{(1 + \alpha z^{-\beta})^m} X(z)$$

(2.13)

The frequency response of the filter is not modified by the transform application. The numerator factor $H(z)(1 + \alpha z^{-\beta})^m$ results in differential coefficients, and the denominator factor $(1+\alpha z^{-\beta})^m$ introduces a recursive accumulator section. DECOR applied to a strength-reduced complex FIR filter is shown in Figure 2.9.

DECOR has been previously shown to reduce transition activity and power consumption
Table 2.3: $\alpha$ and $\beta$ for different types of FIR filters.

<table>
<thead>
<tr>
<th>Filter type</th>
<th>$\alpha$</th>
<th>$\beta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low-pass</td>
<td>$-1$</td>
<td>$1$</td>
</tr>
<tr>
<td>High-pass</td>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>Band-pass</td>
<td>$1$</td>
<td>$\pi/\omega_c$</td>
</tr>
<tr>
<td>Band-stop</td>
<td>$-1$</td>
<td>$2$</td>
</tr>
</tbody>
</table>

Figure 2.9: Decorrelating (DECOR) transform: (a) strength-reduced complex FIR filter architecture, (b) after application of DECOR.
in real FIR filters [11]. DECOR is shown to benefit narrow-band filters the most, where there is a large correlation between adjacent coefficients. DECOR is also effective on complex FIR filters. A cross-coupled complex filter is equivalent to four separate FIR filters, with two adders at the output to compute the final complex result. Two of the FIR portions use the real complex filter coefficients as their own coefficients, and two use the imaginary complex filter coefficients. Thus, the effect of DECOR on a complex filter can be examined by looking at FIR filters with the complex filter's coefficients. Strength-reduced complex filters are constructed similarly, with three FIR filtering sections. Two use the real and imaginary complex filter coefficients as their own coefficients, and the third uses the sum of the real and imaginary coefficients.

By reducing the number of bits required to represent a coefficient, the number of transitions in a multiplier and adder are reduced. Thus, one can examine the reduction in bit width to see the possible power reduction. Figure 2.10 shows the precision reduction for a real and a complex FIR low-pass filter of varying pass-band widths, both with 40 taps and 16-bit fixed-point coefficient. The complex filter has a center frequency of $\pi/10$. For the complex filter, the real, imaginary, and sum of real and imaginary coefficients are separately graphed. The precision reduction is measured from the coefficient that requires the most bits to represent it after applying DECOR. Thus, the bit width of other coefficients may be reduced further.

DECOR also has significant benefits for complex high-pass filters. Figure 2.11 shows the precision reduction for a high-pass complex filter of varying pass-band widths, with a center frequency of $9\pi/10$. However, as the center frequency of the filter moves from zero, the benefits of DECOR reduce. Figure 2.12 shows the precision reduction for a band-pass complex filter of varying pass-band width, with a center frequency of $\pi/5$. Beyond a pass-band width of $0.4\pi$, the same precision is required to fully represent the coefficients.

In [11], a reduction in coefficient bit width of only one bit resulted in 10.5% reduction in transition activity for a 40-tap low-pass FIR filter. Even with no reduction in bit width,
Figure 2.10: Precision reduction after applying DECOR to a 40-tap, 16-bit real low-pass filter and complex low-pass filter (centered at $\omega = \pi/10$).

Figure 2.11: Precision reduction after applying DECOR to a 40-tap, 16-bit complex high-pass filter (centered at $\omega = 9\pi/10$).
transition activity decreased by 8.5%, due to the reduced dynamic range of the coefficients. This suggests that the complex FIR filter and strength-reduced complex FIR filters will also exhibit significant power savings thanks to DECOR.

To apply DECOR to complex filters, the architecture optimizer searches for the last tap in each FIR section. After it, another tap and an accumulator portion are added. Since the filters designed have fixed coefficients, the differences between coefficients are not calculated as part of the architecture. Rather, the coefficients are renamed to represent the differential coefficients.

### 2.3 Voltage Scaling

The architecture optimizer combines voltage scaling with pipelining to generate very low power designs. The propagation delay associated with the charging and discharging of capacitances in the critical path can be written as

$$T_{pd} = \frac{C_{\text{charge}} V_{DD}}{K(V_{DD} - V_t)\alpha}$$

(2.14)
where $C_{\text{charge}}$ is the capacitance along the critical path, $V_{DD}$ is the supply voltage, $V_t$ is the threshold voltage, and $K$ and $\alpha$ are functions of the technology parameters, with $\alpha$ ranging from one to two. Equation (2.14) shows that lowering the supply voltage will increase the propagation delay. The power consumption of the filter can be estimated by

$$P = C_{\text{total}}V_{DD}^2 f \quad (2.15)$$

where $C_{\text{total}}$ is the total capacitance of the circuit and $f$ is the clock frequency of the circuit. From Equation (2.15), decreasing $V_{DD}$ results in a quadratic reduction in power. By first pipelining the circuit to achieve $T_{pd}$ smaller than required, the supply voltage can then be lowered until Equation (2.14) gives the desired propagation delay, resulting in a significant savings in power consumption [1].

After computing the critical path $T_{cp}$, the architecture optimizer compares this with the user’s delay specification $T_{\text{spec}}$. Using Equation (2.14), the required supply voltage $V'_{DD}$ to operate at delay $T_{\text{spec}}$ is found by solving

$$T_{\text{spec}}V_{DD}(V'_{DD} - V_t)^{\alpha} = (V_{DD} - V_t)^{\alpha}V'_{DD}T_{cp} \quad (2.16)$$

for $V'_{DD}$.

### 2.4 Optimizer Output

The architecture optimizer produces a DFG of the final architecture, with embedded information on precisions and delays. This is provided to the fluid cell optimizer, which adds the sizes of the adder and multiplier full adders to the graph as described in Chapter 3 and exports it to the layout synthesizer. The fluid cell optimizer populates the AlphaCArray and AlphaSArray vertex properties with arrays listing the sizes of every full adder cell in each adder and multiplier.
Figure 2.13: Conversion of DFG to HDFG: (a) data-flow graph, (b) equivalent hardware data-flow graph (HDFG).

The DFG is textually represented by a listing of nodes. With each node are the properties of that node, followed by a list of edges leading out of the node and their properties, and a list of edges leading into the node and their properties. The full adder cell sizes are output to a second output file, which lists the vertex number followed by the array of cell sizes.

Additionally, the optimizer generates VHDL output that can be synthesized. The VHDL description is created from instances of behavioral descriptions of multipliers, adders, and latches. These components are parameterized by precision. First, the DFG, which contains delays attached to edges, is converted to a hardware data-flow graph (HDFG), where these delays are converted to nodes and every node corresponds to a physical hardware unit. An example DFG and its corresponding HDFG are shown in Figure 2.13. Next, a number is assigned to each node and edge. For every edge, a signal is created with the correct precision, indexed by the edge number. For each node, the appropriate hardware is instantiated in the VHDL description, with the corresponding edge numbers as inputs and outputs. The optimizer can also create a VHDL file with the actual hardware unit delays, as determined by the delay models. While not appropriate for synthesizing, this model can be used for simulation.
CHAPTER 3

PIPELINE STAGE OPTIMIZATION VIA DELAY BALANCING

Cell-based designs, such as synthesis and the fluid IP core generator approaches, allow flexible designs with a short design cycle. However, traditional cell libraries have fixed transistor sizes, causing unnecessarily high power consumption. The optimization of transistor sizes present in full custom designs is missing from cell-based approaches.

A fluid cell library, where parameterized cells contain groups of scalable transistors, allows the power reductions possible in custom designs to be achieved. A fluid cell library allows designers to choose the most appropriate transistor sizes; noncritical paths can use minimum sized cells, while delay-critical portions use cells with larger transistor sizes and shorter delays.

Using a fluid cell library inside an automated design framework such as the fluid IP core generator requires additional pieces, however. Algorithms to determine scaling factors are needed. Also, these algorithms need accurate models of cell delay, parameterized by the scaling factors. Tradeoffs exist between the scaling flexibility and model complexity. A cell with every transistor individually scalable requires a prohibitively complex model, while an easily modeled cell with a single scale factor lacks the flexibility that a cell library with multiple drive strengths might have.

Furthermore, interconnect capacitance makes up a significant portion of delay in deep submicron circuits. The design approach must account for this delay, through increased com-
plexity in models, or layout techniques that produce minimal and predictable interconnect structures.

Previous work exists on techniques to size transistors and cells as part of an automated design cycle. An approach to transistor sizing in a module-based design is presented in [17]. In this approach, leaf cell layouts are automatically generated with varying transistor sizes, and nonlinear programming is used to size transistors to equalize path lengths. However, empty space, or slack, is initially provided in the layout between rows of leaf cells to accommodate different transistor sizes. Thus, transistor sizes are limited by the availability of slack, and in areas with smaller transistors empty space results. A similar approach uses postlayout transistor sizing for power reduction [18]. After initial cell placement, algorithms are used to downsize transistors, modeling cell delays as functions of load and transistor sizes using look-up tables. This method again results in area-inefficient designs: to allow interconnect to remain unchanged, maximum sized cells are used, regardless of the internal transistor sizes. Furthermore, the initial layout still relies on a method similar to traditional synthesis, resulting in unpredictable layouts with extensive interconnect.

The framework of the core generator provides several advantages over these works. Because the core generator designs specific blocks, specific architectural information can be used to optimally size cells. Additionally, the predictable architectures allow for simple, accurate delay modeling at the architecture level. Layout synthesis ensures regular layouts without a large amount of empty space from the scaling of transistor sizes. Layout techniques also produce minimal and predictable interconnect structures, allowing for simple modeling.

The core generator uses architecture-aware delay balancing algorithms to size the cells in filters. The DSP structures created by the fluid core generator consist almost entirely of multipliers and between pipeline stages. Since the core generator knows the structure of the adders and multipliers, costly optimization algorithms can be avoided. Algorithms for optimizing the specific adder and multiplier architectures suffice.

The adder and multiplier architectures are dominated by the full adder, and thus consid-
erable attention is given to its design and modeling, as discussed in Section 3.1. Algorithms
for delay balancing in multipliers and adders are discussed in Section 3.2.

3.1 Fluid Full Adder Cell

3.1.1 Architecture

The fluid core generator uses a CMOS full adder cell with the schematic shown in Figure
3.1. The CMOS design used is inherently low-power. The cell has a two-stage design, with
the first stage generating the carry output and the second generating the sum output. This
two-stage approach facilitates optimization, as the second stage delay will not affect that of
the first stage (except for a negligible effect on the load capacitance seen by the first stage).

Each stage of the full adder cell has a single scale factor: $\alpha_c$ for the first stage, which
generates the carry output, and $\alpha_s$ for the second stage, which generates the sum output.
When $\alpha_c$ and $\alpha_s$ are both one, all of the NMOS transistors are minimum sized, and the PMOS
transistors are sized such that the propagation delay for a rising and falling transition are equal. Both \( \alpha_c \) and \( \alpha_s \) can be increased from 1 in increments of 1/6 to 5, where the transistors are five times the size as those with \( \alpha \) values of one. The increment value is a limitation imposed by the grid requirements of the design rules used.

For full adder cells in some structures, such as those in the carry chain of a ripple-carry adder, the delay of the second stage is inconsequential, as the critical path is through the carry output. For these cases, it is beneficial to allow both of the PMOS and NMOS transistors in the second stage to be minimum sized to reduce power consumption. The rising and falling transition propagation delays will be mismatched, but this is unimportant outside the critical path. To accomplish this, \( \alpha_s \) can be equal to the special case value of \( \text{min} \). At the layout level, this requires a total of four scaling parameters: \( \alpha_c \), which sizes all of the transistors in the first stage, \( \alpha_{sN} \), which sizes the NMOS transistors in the second stage, \( \alpha_{SP1} \), which sizes the second stage PMOS transistors not part of the output buffer, and \( \alpha_{SP2} \), which sizes the PMOS transistors in the second stage output buffer. These groups of transistors are indicated on the full adder cell layout in Figure 3.2. When \( \alpha_s \) is between 1 and 5, the second stage scaling parameters are fixed functions of \( \alpha_s \).

\[
\begin{align*}
\alpha_{sN} &= \alpha_s \quad (3.1) \\
\alpha_{SP1} &= 2 \alpha_s \quad (3.2) \\
\alpha_{SP2} &= 4.5 \alpha_s \quad (3.3)
\end{align*}
\]

When \( \alpha_s \) is \( \text{min} \), \( \alpha_{sN} = \alpha_{SP1} = \alpha_{SP2} = 1 \).

### 3.1.2 Delay model

Models for the delay of the full adder cell were generated from the techniques in [2]. There, the delay through a cell with a single scale factor is represented as a function of
transistor size and capacitance:

\[ T_d(\alpha, C_{load}) = T_i(\alpha_D) + K_{load}(\alpha_L)C_{load}(\alpha_L) \]  

(3.4)

where \( \alpha_D \) is the size of the driving cell, \( \alpha_L \) is the size of the loading cell, \( T_i(\alpha) \) is the empirically determined intrinsic delay of the cell, independent of load capacitance, \( K_{load}(\alpha) \) is the empirically determined slope of the delay-capacitance curve, and \( C_{load}(\alpha) \) is the load capacitance.

Curve fitting techniques are used to determine parameters of Equation (3.4) from empirical data. To directly extend this to a function of several scale factors requires a table lookup method as opposed to a single equation. However, the two-stage design allows the full adder cell to be modeled as two separate cells. To do so, the layout was modified to remove the connections between the two stages, as shown in Figure 3.3. Since the overall layout was not
modified, the parasitic capacitances were accurately modeled. The capacitance of the inputs was modeled separately for each stage, along with the capacitance of the \( C_{out} \) input to the second stage.

The delay and capacitance models follow below.

Stage 1:

\[
T_{pd}(A \rightarrow C_{out}) = (-1.1954 \times 10^{-6} \alpha_c^2 + 0.001853 \alpha_c + 0.17197) + \\
(-0.22583 \alpha_c^3 + 2.6908 \alpha_c^2 - 11.049 \alpha_c + 17.857) C_{load} \quad (3.5)
\]

\[
T_{pd}(B \rightarrow C_{out}) = (-2.6807 \times 10^{-5} \alpha_c^2 + 0.0019523 \alpha_c + 0.20814) + \\
(-0.22699 \alpha_c^3 + 2.7223 \alpha_c^2 - 11.217 \alpha_c + 18.28) C_{load} \quad (3.6)
\]

\[
T_{pd}(C_{in} \rightarrow C_{out}) = (-0.00024994 \alpha_c^2 + 0.00074058 \alpha_c + 0.15661) + \\
(-0.22605 \alpha_c^3 + 2.692 \alpha_c^2 - 11.044 \alpha_c + 17.704) C_{load} \quad (3.7)
\]
Stage 2:

\[ T_{pd}(C_{out} \rightarrow S) = (-0.0022962 \alpha_s^2 + 0.024327 \alpha_s + 0.33145) + \]
\[ (-0.18511 \alpha_s^3 + 2.1812 \alpha_s^2 - 8.8101 \alpha_s + 14.256) C_{load} \]  
(3.8)

Input capacitance:

\[ C_{C_{out}} = 0.2973 \alpha_s^3 - 3.1538 \alpha_s^2 + 11.8189 \alpha_s - 6.9634 \]  
(3.9)

\[ C_{A_1} = 9.5361 \alpha_c - 1.7585 \]  
(3.10)

\[ C_{B_1} = 11.1976 \alpha_c - 4.5784 \]  
(3.11)

\[ C_{C_{in_1}} = 4.1316 \alpha_c - 0.6774 \]  
(3.12)

\[ C_{A_2} = 6.8583 \alpha_s - 0.1535 \]  
(3.13)

\[ C_{B_2} = 6.8943 \alpha_s - 0.2763 \]  
(3.14)

\[ C_{C_{in_2}} = 7.0456 \alpha_s + 0.5508 \]  
(3.15)

To use the models when \( \alpha_s = \text{min} \), the delay and capacitance were measured when the second stage transistors were minimum size. This determined an equivalent value of \( \alpha_s = 0.55 \) for use in the equations. Using this value results in the correct delay and capacitance for a minimum sized second stage.

### 3.2 Delay Balancing

After the core generator produces an architecture, the fluid cell optimizer sizes the cells in the adders and multipliers to reduce power consumption. Each computational unit has an initial critical path, determined by the precisions of the inputs and the scale factor of the full adder cells in the adder or multiplier. Initially, all cells in the adders and multipliers are of the same size. The majority of the paths through the adder or multiplier have a shorter
delay than the critical path, many by a large degree. The \textit{slack} of each path is defined as the difference between the path’s delay and the delay of the critical path. The goal of the fluid cell optimizer is to scale the full adder cells to reduce the slack of the paths outside the critical path by increasing their delay until it is as close to the critical path delay as possible. Delay balancing retains the original throughput of the adder or multiplier, as it does not size the transistors that make up the critical path.

In a complex structure like a multiplier, delay balancing also serves to reduce extra transitions associated with glitching. When signals reach a gate input at different times, the output may be in the wrong state for a period of time until all the inputs settle. If inputs are simultaneously available at the input, the output will only transition once. Since erroneous transitions require charging and discharging of the transistors in the gate, delay balancing helps to reduce power consumption.

Between pipeline stages, there are four structures that can be encountered in a filter from the core generator: an adder, a multiplier, a multiplier followed by an adder, and one stage of a fine-grain pipelined multiplier-adder. Similar techniques are used in all cases, with slightly different applications. In all cases, the critical path of the entire filter is the upper constraint on delay.

\subsection{Ripple-carry adder delay balancing}

The core generator uses ripple-carry adders because of their regularity in layout. Figure 3.4 shows a 5-bit adder, with the critical path highlighted.

For each bit except the MSB, the second stage of the full adder cell is not part of the critical path. Thus, the paths from the inputs to the sum outputs contain slack, and \( \alpha_s \) of the full adder cells can be reduced. Care must be taken, however, since increasing the delay through the second stage of non-MSB bits could increase the delay from an input to that cell’s output beyond the critical path delay. Thus, the fluid cell optimizer uses the following general algorithm to size the full adder cells in an \( N \)-bit adder, where \( \alpha_s \) is \( \alpha_s \) for the \( i \)th
full adder cell and $T_{si}$ is the delay from the input of the LSB to the sum output of the $i$th full adder cell, where $i = 0$ is the LSB and $i = N$ is the MSB.

$$T_{cp} = \text{critical path of adder}$$
for $i = 1$ to $N - 1$
while $T_{si} < T_{cp}$
$$\alpha_{si} = \alpha_{si} - 1/6$$
end while
if $T_{si} > T_{cp}$
$$\alpha_{si} = \alpha_{si} + 1/6$$
end if
end for

The algorithm reduces the size of $\alpha_s$ of each cell except that of the MSB, starting with the LSB. Transistor sizes are reduced until the delay from the input to the sum output of the cell becomes larger than the critical path $T_{cp}$.

For the full adder used by the core generator, it can be shown that setting $\alpha_s$ to $\min$ of every full adder cell except that of the MSB will not increase the critical path of the cell. Let $T_{d_{N-1}}$ be the delay from the input of the LSB full adder to the sum output of the $N - 1$th full adder. The slack between the critical path and $T_{d_{N-1}}$ is

$$\text{slack} = T_{cp} - T_{d_{N-1}}$$
$$= (T_{d_{N-1}} - T_{(N-1)s} + T_N) - T_{d_{N-1}}$$
$$= T_N - T_{(N-1)s}$$

(3.16)

where $T_{(N-1)s}$ is the delay through the second stage of the $N - 1$th full adder, and $T_N$ is the
delay through the $N$th full adder.

Decreasing $\alpha_s$ of the $N-1$th full adder increases the delay through its second stage to $T'_{(N-1)s}$:

$$slack = T_N - T'_{(N-1)s}$$ (3.17)

Thus, the slack will be greater than zero if

$$T_N > T'_{(N-1)s}$$ (3.18)

From the delay models, the minimum value of $T_N$ (assuming a load of a latch with an input capacitance of 20 pF) is when $\alpha_c = \alpha_s = 5$, resulting in $T_N = 0.487$ ns. The maximum value of $T'_{(N-1)s}$ is when $\alpha_s = \text{min}$, resulting in $T'_{(N-1)s} = 0.470$ ns. Thus, regardless of the initial value of $\alpha$ in the $N$-bit adder, $\alpha_s$ can be reduced to $\text{min}$ in all full adders except the MSB without increasing the delay past the original critical path, as shown in Figure 3.5. If a different full adder cell is used, this can also give a bound on the sizing of $\alpha_s$ for different initial $\alpha$ sizes.

### 3.2.2 Baugh-Wooley multiplier delay balancing

The multiplier structure used by the core generator is the Baugh-Wooley multiplier [19]. This structure allows two's-complement multiplication in a regular, repeated structure. Fig-
Figure 3.6: 8 × 12 Baugh-Wooley multiplier, with critical path highlighted and scaling regions indicated.

Figure 3.6 shows a block diagram of an 8 × 12 Baugh-Wooley multiplier, with each box representing a full adder cell. The critical path delay $T_{cp}$ is highlighted.

All paths other than the critical path have delay less than $T_{cp}$. Balancing the path delays reduces cell sizes, which results in lower power consumption. Since there are a multitude of paths through the multiplier and each cell has two different scaling parameters, brute force techniques cannot be used to optimize the cell sizes in an efficient manner. Strategies on cell sizing were developed by observing differences in the paths through different regions of the multiplier. As in the case of the $N$-bit adder, $T_{cp}$ of the Baugh-Wooley multiplier is calculated using the delay models. This is the upper constraint on the delay of any path in the multiplier.

To begin, any path to a multiplier output other than the MSB will have a smaller delay than the critical path. Thus, $\alpha_\delta$ of the cells on the outputs highlighted in Figure 3.6 can be set to minimum size, as this will affect only their sum delay.

Next, the cells are sized in region A, outlined in Figure 3.6. All of the paths through these cells contain fewer full adder delays than the critical path. Any path passing through a cell in column 1 will have at least one fewer full adder delay than the critical path, any
path passing through a cell in column 2 will have at least two fewer full adder delays, and so on. Thus, significant slack exists in the cell of this region, and the transistor sizes can be decreased until the delay through these paths is equal to $T_{cp}$. To accomplish this, the optimizer starts with the cell labeled 1. First, $\alpha_s$ is reduced until a path from the multiplier inputs to the MSB output, passing through cell 1, is greater than $T_{cp}$, or until $\alpha_s$ is minimum size. If $\alpha_s$ is reduced to minimum size, $\alpha_e$ is reduced under the same constraints. The cells are sized in the order indicated in Figure 3.6. With this approach, sizing of the current cell does not affect the paths through cells already sized.

The final step is to size the cells in region B indicated in Figure 3.6. A path starting at an input at the top of this region and ending at the MSB output has the same number of full adder delays as the critical path. However, paths starting in this region will contain a larger proportion of carry delays to sum delays than the critical path. Paths passing through the cells in column 1 will trade at least one sum delay for a carry delay, paths passing through the cells in column 2 will trade at least two sum delays for carry delays, and so forth. Since a path to the carry output can have as little as half the delay of a path to the sum output in the two-stage full adder design, there is significant slack to be exploited.

The cells are sized in the order indicated in Figure 3.6. This order is chosen for two reasons. First, the sizing of earlier cells does not reduce the slack in the paths through some of the later cells to be sized. For example, sizing cell 1 does not affect the delay paths through cell 2, sizing cell 3 does not affect the delay of paths through cells 4 and 5, and so on. This evenly distributes the slack throughout the region. In contrast, if cell 15 was first sized the paths through every other cell in this region would increase in delay. Second, sizing the cells in column 1 would create a barrier between those cells and the cells in the critical path. Since full adders in the critical path are not scaled, those cells will be the largest of any in the multiplier. Greatly reducing the size of cells in column 1 results in small cells driving large loads. The delay is much larger than if a tapered buffer approach is used, where the size of cells is gradually increased from small to large along the path. By starting with cell
one and continuing to the left, a natural taper is built, leading to the adders in the critical path.

Figure 3.7 shows the how the cells of an $N \times M$ Baugh-Wooley multiplier, $N > M$, are referenced as an array in the delay balancing algorithm. The algorithm for balancing the delay by sizing the cells as described above appears in Figure 3.8.

Figure 3.9 shows the final cell sizes and Figure 3.10 shows the layout for a $6 \times 10$ multiplier before and after fluid cell optimization, with an initial full adder cell scaling of five times minimum size. The layouts are generated using the layout synthesis techniques presented in [7].

3.2.3 Unpipelined multiplier-adder delay balancing

Figure 3.11 shows a block diagram of a multiplier followed by an adder. Here, multiple critical paths exist, as highlighted in the figure. The lower bits of the adder are a natural extension to region B of the Baugh-Wooley multiplier. With this addition to region B, the same algorithm as in Section 3.2.2 is used, with the following modification: $\alpha_{x2} = \min$ for the adder cells other than the MSB, instead of for the output cells of the multiplier. Also, all paths through cells are measured from the multiplier inputs to the output of the MSB of the adder. Figure 3.11 shows the changes to Region B, with the order of cell sizing noted.
\( T_{cp} \) = critical path of multiplier 
\( \alpha_{s}[N, 2 \ldots M - 1] = \min; \alpha_{s}[1 \ldots N, M - 1] = \min \) 
for \( i = 1 \) to \# of cells in region A 
\( T_{di} = \max T_{d} \forall \) paths containing cell \( i \) 
while \( T_{d} < T_{cp} \) 
\( \alpha_{si} = \alpha_{si} - 1/6 \) 
\( T_{d} = \max T_{d} \forall \) paths containing cell \( i \) 
end while 
if \( T_{d} > T_{cp} \) 
\( \alpha_{si} = \alpha_{si} + 1/6 \) 
else 
if \( \alpha_{si} = 1.0 \) 
\( \alpha_{si} = \min \) 
\( T_{d} = \max T_{d} \forall \) paths containing cell \( i \) 
if \( T_{d} > T_{cp} \) 
\( \alpha_{si} = 1.0 \) 
end if 
end if 
while \( T_{d} < T_{cp} \) 
\( \alpha_{ci} = \alpha_{ci} - 1/6 \) 
\( T_{d} = \max T_{d} \forall \) paths containing cell \( i \) 
end while 
if \( T_{d} > T_{cp} \) 
\( \alpha_{si} = \alpha_{ci} + 1/6 \) 
end if 
end if 
end for 
for \( i = 1 \) to \# of cells in region B 
\( T_{d} = \max T_{d} \forall \) paths containing cell \( i \) 
while \( T_{d} < T_{cp} \) 
\( \alpha_{s} = \alpha_{s} - 1/6 \) 
\( T_{d} = \max T_{d} \forall \) paths containing cell \( i \) 
end while 
if \( T_{d} > T_{cp} \) 
\( \alpha_{s} = \alpha_{s} + 1/6 \) 
else 
if \( \alpha_{s} = 1.0 \) 
\( \alpha_{s} = \min \) 
\( T_{d} = \max T_{d} \forall \) paths containing cell \( i \) 
if \( T_{d} > T_{cp} \) 
\( \alpha_{s} = 1.0 \) 
end if 
end if 
while \( T_{d} < T_{cp} \) 
\( \alpha_{c} = \alpha_{c} - 1/6 \) 
\( T_{d} = \max T_{d} \forall \) paths containing cell \( i \) 
end while 
if \( T_{d} > T_{cp} \) 
\( \alpha_{s} = \alpha_{c} + 1/6 \) 
end if 
end if 
end for 

Figure 3.8: Algorithm for sizing cells in a Baugh-Wooley multiplier.
Figure 3.9: Baugh-Wooley multiplier scaling results.

Figure 3.10: Layout of Baugh-Wooley multiplier: (a) before scaling, (b) after scaling.
3.2.4 Pipelined multiplier-adder delay balancing

When the multiplier-adder is fine-grain pipelined into two stages, the appropriate cutset must be found to equalize the delay of each stage. Since the delays of the different paths through a full adder vary, the cutset cannot be determined by simply putting the same number of full adder cells on each side. Instead, the difference in sum and carry delays must be noted.

Figure 3.12 shows a $10 \times 12$ Baugh-Wooley multiplier followed by a 23-bit adder, with an initial $\alpha$ of three for each cell. The appropriate cutset and the critical path of each stage are indicated. Since a carry delay is shorter than a sum delay, the critical path of the first stage passes through fewer full adders than the critical path of the second stage to equalize the delays. Since the size of the adder is partially dependent on the number of taps in the filter, different filters may require different cutsets even if the multiplier precision remains the same. A portion of the cutset may move as indicated in Figure 3.12 to equalize the delay in each stage. Moving the cutset to the left removes a carry delay from the second stage, and adds a carry delay to the first stage. Moving it to the rightmost position in Figure 3.12 removes two sum delays from the first stage while adding a single carry delay to the second stage.

To scale the cells in the first stage, the same algorithm as in Section 3.2.3 is used. The
paths through the cells are measured from the multiplier inputs to the latches at the end of the pipeline stage. In the second stage there exist many critical paths with identical delays, as shown in Figure 3.12. Since a critical path passes through every path in the full adders in the second-to-last row, no scaling can be performed here. The full adders in the last row can be dealt with as an $N$-bit adder.

### 3.3 Results

Simulations were used to demonstrate the power savings from delay balancing. Layouts were created by the layout synthesizer, within the Cadence Virtuoso layout editor. Netlists were extracted with parasitic capacitances, and power consumption was determined from SPICE simulations.

Table 3.1 summarizes the power savings for different sized adders. The starting $\alpha$ is the initial size of every full adder before cell scaling is performed. Of note is the power reduction when the starting $\alpha$ value is one. These savings are achieved from sizing the PMOS second stage transistors to minimum size. An approach using a nonfluid cell library could not size transistors in this manner; even with a minimum drive strength cell, transistors are sized to
<table>
<thead>
<tr>
<th>Adder size</th>
<th>Delay (ns)</th>
<th>Starting $\alpha$</th>
<th>Power - before scaling ($\mu$W)</th>
<th>Power - after scaling ($\mu$W)</th>
<th>Power reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>1.39</td>
<td>5</td>
<td>437</td>
<td>326</td>
<td>25.3%</td>
</tr>
<tr>
<td>4 bits</td>
<td>1.44</td>
<td>3</td>
<td>269</td>
<td>214</td>
<td>20.4%</td>
</tr>
<tr>
<td>4 bits</td>
<td>1.65</td>
<td>1</td>
<td>95</td>
<td>79</td>
<td>17.5%</td>
</tr>
<tr>
<td>16 bits</td>
<td>4.99</td>
<td>5</td>
<td>829</td>
<td>518</td>
<td>37.5%</td>
</tr>
<tr>
<td>16 bits</td>
<td>5.53</td>
<td>1</td>
<td>197</td>
<td>159</td>
<td>19.4%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mult. size</th>
<th>Delay (ns)</th>
<th>Starting $\alpha$</th>
<th>Power - before scaling (mW)</th>
<th>Power - after scaling (mW)</th>
<th>Power reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$16 \times 16$</td>
<td>13.07</td>
<td>5</td>
<td>17.37</td>
<td>11.54</td>
<td>33.6%</td>
</tr>
<tr>
<td>$16 \times 16$</td>
<td>13.63</td>
<td>2</td>
<td>7.30</td>
<td>5.12</td>
<td>29.5%</td>
</tr>
<tr>
<td>$16 \times 16$</td>
<td>14.08</td>
<td>1</td>
<td>3.87</td>
<td>3.02</td>
<td>21.9%</td>
</tr>
<tr>
<td>$10 \times 16$</td>
<td>9.37</td>
<td>3</td>
<td>5.37</td>
<td>3.84</td>
<td>28.5%</td>
</tr>
<tr>
<td>$10 \times 16$</td>
<td>10.14</td>
<td>1</td>
<td>1.95</td>
<td>1.64</td>
<td>15.9%</td>
</tr>
<tr>
<td>$8 \times 12$</td>
<td>7.18</td>
<td>3</td>
<td>2.58</td>
<td>1.95</td>
<td>24.4%</td>
</tr>
<tr>
<td>$8 \times 12$</td>
<td>7.85</td>
<td>1</td>
<td>1.29</td>
<td>1.06</td>
<td>17.8%</td>
</tr>
</tbody>
</table>

equalize rising and falling propagation delays.

In Table 3.2, results of multiplier delay balancing for several different sizes of multipliers are presented. Starting $\alpha$ is the initial size of $\alpha_c$ and $\alpha_s$ for every full adder before the scaling algorithm is applied. As in the case of the $N$-bit adder, significant power savings are achievable even with an initial $\alpha$ of one. Also, $\alpha$ can be increased from 1 to 5 with only a $3.5x$ increase in power consumption after scaling.

Table 3.3 gives the results of delay balancing for unpipelined taps, and Table 3.4 gives the results of delay balancing for two-stage fine-grain pipelined taps. Again, power savings are achievable even with an initial $\alpha$ of one. The smaller power savings compared to individual multipliers and adders can be attributed to the additional overhead of the latches present at the tap level.

Table 3.5 shows corresponding results for synthesized multipliers. The standard cell library used contained 43 cells, consisting of logic gates at multiple drive strengths, several
Table 3.3: Unpipelined tap delay balancing results.

<table>
<thead>
<tr>
<th>Mult. size</th>
<th>Adder size</th>
<th>Delay (ns)</th>
<th>Starting $\alpha$</th>
<th>Power - before scaling (mW)</th>
<th>Power - after scaling (mW)</th>
<th>Power reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$16 \times 16$</td>
<td>35</td>
<td>15.63</td>
<td>3</td>
<td>11.1</td>
<td>8.30</td>
<td>25.2%</td>
</tr>
<tr>
<td>$16 \times 16$</td>
<td>35</td>
<td>16.08</td>
<td>1</td>
<td>4.35</td>
<td>3.61</td>
<td>17.0%</td>
</tr>
<tr>
<td>$10 \times 16$</td>
<td>29</td>
<td>11.37</td>
<td>3</td>
<td>7.28</td>
<td>5.40</td>
<td>25.8%</td>
</tr>
<tr>
<td>$10 \times 16$</td>
<td>29</td>
<td>12.14</td>
<td>1</td>
<td>2.90</td>
<td>2.56</td>
<td>11.7%</td>
</tr>
<tr>
<td>$8 \times 12$</td>
<td>23</td>
<td>9.18</td>
<td>3</td>
<td>4.79</td>
<td>3.57</td>
<td>25.5%</td>
</tr>
<tr>
<td>$8 \times 12$</td>
<td>23</td>
<td>9.85</td>
<td>1</td>
<td>1.95</td>
<td>1.76</td>
<td>9.8%</td>
</tr>
</tbody>
</table>

Table 3.4: Fine-grain pipelined tap delay balancing results.

<table>
<thead>
<tr>
<th>Mult. size</th>
<th>Adder size</th>
<th>Delay (ns)</th>
<th>Starting $\alpha$</th>
<th>Power - before scaling (mW)</th>
<th>Power - after scaling (mW)</th>
<th>Power reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$16 \times 16$</td>
<td>35</td>
<td>7.76</td>
<td>3</td>
<td>10.2</td>
<td>7.59</td>
<td>25.5%</td>
</tr>
<tr>
<td>$16 \times 16$</td>
<td>35</td>
<td>8.43</td>
<td>1</td>
<td>4.36</td>
<td>3.72</td>
<td>14.6%</td>
</tr>
<tr>
<td>$10 \times 16$</td>
<td>29</td>
<td>6.01</td>
<td>3</td>
<td>7.16</td>
<td>4.43</td>
<td>38.1%</td>
</tr>
<tr>
<td>$10 \times 16$</td>
<td>29</td>
<td>6.87</td>
<td>1</td>
<td>3.38</td>
<td>2.92</td>
<td>13.6%</td>
</tr>
<tr>
<td>$8 \times 12$</td>
<td>23</td>
<td>4.98</td>
<td>3</td>
<td>4.85</td>
<td>3.19</td>
<td>34.2%</td>
</tr>
<tr>
<td>$8 \times 12$</td>
<td>23</td>
<td>5.12</td>
<td>1</td>
<td>2.19</td>
<td>2.02</td>
<td>7.8%</td>
</tr>
</tbody>
</table>

flip-flops. and two full adder cells: a low-speed, low-power static logic cell, and a high-speed, high-power pass-transistor logic cell. Synthesis was performed with the Synopsys Design Compiler. For both cases, synthesis consumed more than twice the power of the core generator multiplier.

Table 3.5: Synthesis multiplier power comparison.

<table>
<thead>
<tr>
<th>Mult. size</th>
<th>Delay (ns)</th>
<th>Power (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$16 \times 16$</td>
<td>17.23</td>
<td>6.52</td>
</tr>
<tr>
<td>$8 \times 12$</td>
<td>10.46</td>
<td>2.14</td>
</tr>
</tbody>
</table>

44
CHAPTER 4

TEST CHIP

To demonstrate the benefits of the fluid IP core generator, three complex FIR filters were generated with the core generator and compared to the traditional synthesis approach. A low-speed and a high-speed filter were targeted, both employing transforms and cell scaling to generate low-power designs. In addition, a low-speed filter without fluid cells was designed, allowing the added benefits of using fluid cells to be measured.

The filters were developed for a TSMC 0.25-\(\mu\)m, 2.5-V CMOS process. Simulation results highlight the power savings from the fluid IP core generator approach. A test chip with the four filters has been submitted for fabrication.

4.1 Filter Descriptions

All four filters were five-tap complex FIR filters, with 8-bit input precision, 12-bit coefficient precision, and 23-bit accumulator precision. This is comparable in size and precision to filters used in current read channel applications [20],[21],[22], large enough to demonstrate the benefits of transform templates and cell scaling, and still small enough to fit several designs onto one chip. All of the filters were designed with programmable coefficients, but the coefficients were set to fixed values to ease final chip testing. A band-pass filter was implemented with the frequency and phase response shown in Figure 4.1 and the transfer
Figure 4.1: Frequency and phase response of test chip filters.

The low-speed filters had a target throughput of 20 MHz. Two core generator filters were designed at this speed; both incorporated architectural transforms and voltage scaling.
one with fixed cells and one using fluid cells. A synthesis filter running at 20 MHz was also included for comparison. All three filters employed voltage scaling to further reduce power consumption.

A single core generator filter with a throughput of 200 MHz was generated. This design used transform templates to meet the increased throughput requirement, and fluid cells to lower the power. Traditional synthesis could not meet the 200-MHz throughput requirement; thus no high speed synthesis comparison was included on the test chip.

4.1.1 Fluid IP core generator filters

For the low-speed filters, the core generator used a transpose, strength-reduced initial architecture. This data-broadcast architecture is naturally pipelined to the tap level, with a critical path of a multiplier followed by an adder. Since this filter is not speed-critical, the initial full adder cell size was chosen to be as small as possible, with $\alpha = 1$. Also, this cell size is equivalent to a standard cell with a drive of 1X, allowing a fair comparison with synthesis. The fixed-cell filter used full adders exclusively of this size, while the fluid cell filter uses the optimization techniques in Chapter 3 to further reduce power.

The high-speed filters start with the same initial transpose, strength-reduced complex FIR architecture. An additional level of pipelining is added, which is used to fine-grain pipeline the multiply-accumulate structures. Using full adder cells with a scale factor of one, the target critical path of 5 ns cannot be met. Thus, the cell size is increased to a scale factor of three, where the modeled critical path of the fine-grain pipelined filter is 4.8 ns.

The fluid core optimizer architectures were input to the layout synthesizer. Scripts written in the Skill programming language within the Cadence environment generated layouts in the Virtuoso layout editor. The multiplier and adder precisions, cell sizes, and pipelining levels were used as inputs to generate individual tap layouts. These were combined to form FIR sections of the filters. The adders at the inputs and outputs of the filter were placed and routed by hand.
To determine the size of the power rails, SPICE simulation was used to estimate the average current drawn by each filter. In the layouts, the power and ground lines are in metal layer 1. From the process design rules, the maximum current density of metal 1 is 1 mA/μm at 110°C [23]. Using this data, the power and ground rails to each filter were sized to have a total width of 1 μm per 1 mA of current drawn.

The clock was routed with the strategy shown in Figure 4.2. From the clock pad, a buffer of drive strength 3 drives the four filters. At each core generator filter, another strength 3 buffer drives the internal clock signals. To minimize clock skew, the clock is routed in the opposite direction to that of the data. Clock jitter is minimized by driving each clock line in the filter by several buffers across the length of the filter. Internally, buffers of drive strength 2 are used. In the slow filters, each tap contains 23 latches, and thus each buffer drives at most 23 latches. The fast filter contains twice as many latches per tap; here, two drive strength 2 buffers are used per tap.
4.1.2 Synthesis benchmark

The benchmark filter was synthesized from a behavioral VHDL description of a transpose form complex FIR filter, included in Appendix B. The standard cell library used contained 43 cells, consisting of logic gates at multiple drive strengths, several flip-flops, and two full adder cells: a low-speed, low-power static logic cell, and a high-speed, high-power pass-transistor logic cell.

Synopsys Design Compiler synthesized the design. Area critical optimization was employed, with Boolean logic optimization enabled and a target clock frequency of 20 MHz. Gate fanout was limited to 5 to reduce interconnect and loading capacitance, and a load of 2 pF was placed at the output pins to ensure sufficient buffering of the outputs. The synthesis output met the critical path requirements, with a reported slack of 38 ns. A Verilog description and timing information for every net was output.

The Cadence Silicon Ensemble tool performed timing-driven placement and routing. The power and ground rings were sized using the same technique as the core generator filters, using SPICE simulations to estimate the average current. Ring widths of 12.5 μm proved to be sufficient. The cells were placed with timing-driven placement, with space reserved for clock buffers. The clock tree was generated to have a skew of less than 100 ps after insertion of clock buffers. Finally, the Warp Route command provided timing-aware routing. The final design was imported into the Virtuoso layout editor.

4.2 Chip Architecture

An overview of the chip architecture is shown in Figure 4.3. The eight real and eight imaginary data inputs are routed from the input pads to latches at the inputs of each of the four filters. There are also latches at the output of each filter, followed by buffers. The eight most significant bits of the filter’s real and imaginary outputs are routed to 4-to-1 multiplexers at the output pads, where the output of a particular filter can be selected via
two select inputs. The clock is buffered at its input pad, and routed to each filter where another buffer drives the internal clock signals.

Each filter has its own power and ground pins, which supply the filter and the latches surrounding it. A separate supply ring for the pad ring is powered by two power and two ground pins. The independent power supplies allow for the power consumption of each filter to be accurately measured from the supply current.

The complete chip layout appears in Figure 4.4. Inputs and filter power supply pads appear on the left, and the output and clock pads appear on the right. Output select and pad ring power supplies are on the top and bottom of the layout.

### 4.3 Results

Table 4.1 summarizes the simulation results from the test chip filters. In all cases, the core generator produced lower-power filters with less area than synthesis. For the low-speed filters, voltage scaling was employed until incorrect outputs were produced by the filters.
Figure 4.4: Layout of fluid IP core generator test chip.
Table 4.1: Simulation results from test chip filters.

<table>
<thead>
<tr>
<th>Design Methodology</th>
<th>Speed</th>
<th>Power Consumption</th>
<th>Area</th>
<th>Power-Delay Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core generated slow filter fixed cells</td>
<td>20 MHz</td>
<td>0.95 mW at 0.98 V</td>
<td>1.12 mm²</td>
<td>47.5 pW-s</td>
</tr>
<tr>
<td>Core generated slow filter fluid cells</td>
<td>20 MHz</td>
<td>0.90 mW at 0.98 V</td>
<td>1.11 mm²</td>
<td>45.0 pW-s</td>
</tr>
<tr>
<td>Core generated fast filter</td>
<td>160 MHz</td>
<td>109.8 mW at 2.5 V</td>
<td>1.47 mm²</td>
<td>686.3 pW-s</td>
</tr>
<tr>
<td>Synthesized slow filter</td>
<td>20 MHz</td>
<td>6.47 mW at 1.5 V</td>
<td>1.59 mm²</td>
<td>323.5 pW-s</td>
</tr>
<tr>
<td>Synthesized fast filter</td>
<td>143 MHz</td>
<td>246.5 mW at 2.5 V</td>
<td>2.68 mm²</td>
<td>1725.5 pW-s</td>
</tr>
</tbody>
</table>

These results show that the core-generated slow filters offer a 7X improvement in energy-efficiency over the corresponding synthesized filter, with a 30% reduction in area. Included in the simulation results is the best effort of synthesis to produce a 200-MHz filter. This high-speed synthesis filter produces incorrect outputs for some of the bits due to timing violations, but is included to provide some comparison for the high-speed core generator filter. In the fast filter case, the core generated filter provides a 2.5X improvement over the synthesized filter along with an area reduction of 45%.

The use of fluid cells and pipeline stage optimization provided an additional 5.3% power reduction beyond what voltage scaling and architectural transforms provided. This is notable, especially since the initial full adder cell size had a scaling factor of one. This means the reduction in power is from sizing PMOS transistors to minimum size in the sum paths of the full adder cells, and would not be achievable from any nonfluid cell-based approach.

4.4 Summary

The fluid IP core generator methodology is shown to provide significant performance benefits over a traditional synthesis design flow. Improvements from both architecture optimization and pipeline stage optimization demonstrate the strength of the core generator approach. The fluid core optimizer transitions seamlessly to new processes and technologies with the development of updated models and layout synthesis techniques. As the core
generator technology matures, new cores, cells, and transformations can be added to the framework, allowing for rapid system-on-a-chip design in the face of the next generation of design challenges.
APPENDIX A

CORE OPTIMIZER SOURCE CODE OUTLINE

This appendix provides an outline of the C++ source code of the fluid core optimizer. Code for the main function is given, with descriptions of each included header file and called function. The Boost Graph Library [24] is used for the graph data structures and algorithms.

//******************************************************************************
// Fluid IP Core Generator
//   cfir.cpp
//   Main function for Complex FIR creation & transformation
//
// Jeff Geib (jgeib@ieee.org)
// Version 1.0, 5/10/02
//******************************************************************************

#include <boost/config.hpp>
#include <iostream>
#include <vector>
#include <fstream.h>
#include <stdlib.h>

#include <boost/utility.hpp>
#include <boost/property_map.hpp>
#include <boost/graph/adjacency_list.hpp>

using namespace boost;
using namespace std;

#include "properties.hpp" // Graph data type description
#include "critical_path.hpp" // Critical path calculation
#include "cfirpipeline.hpp" // Pipelining transform

54
#include "delay.hpp" // Delay models
#include "misc.hpp"  // Utility routines
#include "alphascale.hpp" // Cell scaling algorithms
#include "vhdloutput.hpp" // Creates VHDL description
#include "makefir.hpp"  // Architecture generation
#include "cfirdecor.hpp" // DECOR transform
#include "vscale.hpp"   // Voltage scaling calculations

int main(int argc, char *argv[]) {
  // Process command line parameters
  if (argc < 9) {
    cout << "Usage: " << argv[0] << " <num of taps> <input precision>
         <coefficient precision> <start alpha> <Transpose> <SR> <DECOR>
         <Pipelining> [target]" << endl;
    return(1);
  }

  // # of taps
  int N = atoi(argv[1]);
  // Input precision
  int Pi = atoi(argv[2]);
  // Coefficient precision
  int Pc = atoi(argv[3]);
  // Necessary adder precision to insure no overflow
  int Padder = (int) ceil(Pi+Pc+log(N)/log(2));
  // Starting alpha for all cells
  float startalpha = atof(argv[4]);

  // Transform flags
  int Transpose = atoi(argv[5]);
  int SR = atoi(argv[6]);
  int DECOR = atoi(argv[7]);
  int Pipelining = atoi(argv[8]);

  // target: target critical path (in ns). If specified, the required
  // supply voltage will later be output that provides this exact delay
  float target;
  if (argc == 10)
    target = atof(argv[9]);
  else
    target = 0;

  // Define Graph
MyGraph G;

// Call function to make initial DFG of filter
if (SR) {
    G=CFIR_SR(N, Pi, Pc, Padder, Transpose, startalpha);
} else {
    G=CFIR(N, Pi, Pc, Padder, Transpose, startalpha);
}

// Apply transforms
if (DECOR)
    CFIRDECOR(G, N, Pi, Pc, Padder, startalpha);
if (Pipelining == 1)
    PipelineAdders(Pipelining,G);
if (Pipelining > 1) {
    PipelineAdders(Pipelining,G);
    PipelineMultipliers(1,G);
}

// Fill in the delays of all of the vertices
PopulateDelays(G);
// Scale the alphas in the multipliers
ScaleMultAlphas(G);
// Do the same for the adders
ScaleAdderAlphas(G);

/***********************
// Write output files
/***********************
// Write the a1, a2 arrays for every vertex (for now, called cfir.alphas)
WriteVertexAlphas(G);
// Write a description of the DFG to a file (for now, called cfir.graph)
WriteNetwork(G);

// Calculate the critical path (can take _long_ for large graphs),
// and print it to the screen
float tcp;
 tcp = CriticalPath<MyGraph>(G);
 cout << "Critical path: " << tcp << endl;

// If a target critical path was specified, output required supply scaling
// to meet that speed
if (target > 0)
    cout << "Vdd: " << vscale(2.0,tcp,target)*2.5 << endl;
// Write the VHDL description (for now, called cfir.vhd)
WriteVHDLFile("cfir.vhd",G);

WriteMODFile("cfir.ckt",G);

// Exit w/o error!!
return 0;
}
APPENDIX B

VHDL SOURCE CODE FOR TEST CHIP

This appendix contains the VHDL source code of the synthesis filter included on the test chip.

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

entity DFF is
  generic (Bits: integer);
  port (D: in signed(Bits-1 downto 0);
      CLK: in std_logic;
      Q: out signed(Bits-1 downto 0));
end DFF;

architecture behavioral of DFF is
begin
process (CLK)
  begin
    if (CLK'event and CLK = '1') then
      Q <= D;
    end if;
  end process;
end behavioral;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

entity mult8x12 is
port(ina: in signed(7 downto 0);
inb: in signed(11 downto 0);
f: out signed(19 downto 0));
end mult8x12;

architecture behavioral of mult8x12 is
begin
  f <= ina * inb;
end behavioral;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

entity adder23 is
  port(ina: in signed(19 downto 0);
inb: in signed(22 downto 0);
f: out signed(22 downto 0));
end adder23;

architecture behavioral of adder23 is
begin
  f <= ina + inb;
end behavioral;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

entity adder2323 is
  port(ina: in signed(22 downto 0);
inb: in signed(22 downto 0);
f: out signed(22 downto 0));
end adder2323;

architecture behavioral of adder2323 is
begin
  f <= ina + inb;
end behavioral;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
entity adder8 is
  port(ina: in signed(7 downto 0);
       inb: in signed(7 downto 0);
       f:  out signed(7 downto 0));
end adder8;

architecture behavioral of adder8 is
begin
  f <= ina + inb;
end behavioral;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

entity tap is
  port(x_in: in signed(7 downto 0);
       c_in: in signed(11 downto 0);
       a_in: in signed(22 downto 0);
       clk: in std_logic;
       x_out: out signed(7 downto 0);
       a_out: out signed(22 downto 0));
end tap;

architecture behavioral of tap is
  component mult8x12
  port(ina: in signed(7 downto 0);
       inb: in signed(11 downto 0);
       f:  out signed(19 downto 0));
end component;
  component adder23
  port(ina: in signed(19 downto 0);
       inb: in signed(22 downto 0);
       f:  out signed(22 downto 0));
end component;
  signal m_out: signed(19 downto 0);
  signal adder_out: signed(22 downto 0);
begin
  mult: mult8x12 port map (x_in, c_in, m_out);
  adder: adder23 port map (m_out, a_in, adder_out);
  x_out <= x_in;
  delay: process(clk)
  begin
    if clk'event and clk='1' then
a_out <= adder_out;
end if;
end process delay;
end behavioral;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

delimiter entity
entity FIR is
  port(x_in: in signed(7 downto 0);
    c_in1: in signed(11 downto 0);
    c_in2: in signed(11 downto 0);
    c_in3: in signed(11 downto 0);
    c_in4: in signed(11 downto 0);
    c_in5: in signed(11 downto 0);
    clk: in std_logic;
    y_out: out signed(22 downto 0));
end FIR;
end entity;

delimiter architecture
architecture behavioral of FIR is
  component tap
    port(x_in: in signed(7 downto 0);
         c_in: in signed(11 downto 0);
         a_in: in signed(22 downto 0);
         clk: in std_logic;
         x_out: out signed(7 downto 0);
         a_out: out signed(22 downto 0));
  end component;
type arrayx is array(6 downto 2) of signed(7 downto 0);
type arraya is array(5 downto 2) of signed(22 downto 0);
signal x: arrayx;
signal a: arraya;
signal zero: signed(22 downto 0);
begin
  zero <= "00000000000000000000000000";
tap1: tap port map (x_in, c_in5, zero, clk, x(2), a(2));
tap2: tap port map (x(2), c_in4, a(2), clk, x(3), a(3));
tap3: tap port map (x(3), c_in3, a(3), clk, x(4), a(4));
tap4: tap port map (x(4), c_in2, a(4), clk, x(5), a(5));
tap5: tap port map (x(5), c_in1, a(5), clk, x(6), y_out);
end behavioral;
end architecture;
end package;
end package body;
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

entity CFIR is
  port(xr_in: in signed(7 downto 0);
       xi_in: in signed(7 downto 0);
       clk: in std_logic;
       yr_out: out signed(22 downto 0);
       yi_out: out signed(22 downto 0));
end CFIR;

architecture behavioral of CFIR is
  component FIR
    port(x_in: in signed(7 downto 0);
         c_in1: in signed(11 downto 0);
         c_in2: in signed(11 downto 0);
         c_in3: in signed(11 downto 0);
         c_in4: in signed(11 downto 0);
         c_in5: in signed(11 downto 0);
         clk: in std_logic;
         y_out: out signed(22 downto 0));
  end component;

  component DFF
    generic (Bits: integer);
    port (D: in signed(Bits-1 downto 0);
          CLK: in std_logic;
          Q: out signed(Bits-1 downto 0));
  end component;

  component adder8
    port(ina: in signed(7 downto 0);
         inb: in signed(7 downto 0);
         f: out signed(7 downto 0));
  end component;

  component adder2323
    port(ina: in signed(22 downto 0);
         inb: in signed(22 downto 0);
         f: out signed(22 downto 0));
  end component;
signal y1, y2, y3, y4, negy3: signed(22 downto 0);
signal c_in1: signed(11 downto 0);
signal c_in2: signed(11 downto 0);
signal c_in3: signed(11 downto 0);
signal c_in4: signed(11 downto 0);
signal c_in5: signed(11 downto 0);
signal d_in1: signed(11 downto 0);
signal d_in2: signed(11 downto 0);
signal d_in3: signed(11 downto 0);
signal d_in4: signed(11 downto 0);
signal d_in5: signed(11 downto 0);
begin
  c_in1 <= "111011110101";
c_in2 <= "000010001101";
c_in3 <= "001000011000";
c_in4 <= "000010001101";
c_in5 <= "111011110101";
  d_in1 <= "111100111110";
d_in2 <= "111001001101";
d_in3 <= "000000000000";
d_in4 <= "000110111011";
d_in5 <= "000011000010";
  FIR1: FIR port map (xr_in, c_in1, c_in2, c_in3, c_in4, c_in5, clk, y1);
  FIR2: FIR port map (xi_in, d_in1, d_in2, d_in3, d_in4, d_in5, clk, y2);
  FIR3: FIR port map (xr_in, d_in1, d_in2, d_in3, d_in4, d_in5, clk, y3);
  FIR4: FIR port map (xi_in, c_in1, c_in2, c_in3, c_in4, c_in5, clk, y4);
  negy3 <= -y3;
  adder3: adder2323 port map(y1, y2, yr_out);
  adder4: adder2323 port map(negy3, y4, yi_out);
end;
REFERENCES


VITA

Jeffrey Alan Geib was born in Dayton, Ohio, in 1978. He was awarded the Bachelor of Science degrees in Electrical Engineering and Computer Science and Engineering from the University of Toledo in 2000. In the fall of 2000, he entered the Master of Science program in Electrical Engineering at the University of Illinois at Urbana-Champaign as a teaching assistant. He accepted a research assistantship with the VLSI Information Processing Systems research group in the fall of 2001. He will join the Tactical Electronic Warfare Division of the U.S. Naval Research Laboratory in Washington, D.C., upon graduation.