Automatic VHDL Model Generation of Parameterized FIR Filters

E. George Walters III[1], John Glossner[2], and Michael J. Schulte[1]

[1] Computer Architecture and Arithmetic Laboratory, Computer Science and Engineering Department, Lehigh University, Bethlehem PA 18015 USA
[2] Sandbridge Technologies, 1 N Lexington Ave, 10th Floor, White Plains NY 10601

Abstract. This paper describes a Java-based tool that automatically generates structural level VHDL models of FIR Filters. Automatic generation of VHDL models allows the designer to rapidly explore the design space and test the impact of parameters on the design. The tool is based on a general purpose computer arithmetic component package developed at Lehigh University and can easily be extended to enable rapid prototyping of other hardware accelerators used in embedded systems. In this paper, we describe the effects of truncated multipliers in FIR filters. We show that a 22.5 % reduction in area can be achieved for a 24-tap filter with 16-bit coefficients, and that the reduction error SNR is only 2.4 dB less than the roundoff error SNR of the same filter with no truncation. Using the techniques presented in this paper, the average reduction error of the filter is several orders of magnitude less than the average reduction error of the individual multipliers.

1 Introduction

The design of hardware accelerators for embedded systems presents many design tradeoffs that are difficult to quantify without bit-accurate simulation and area and delay estimates of competing alternatives. Structural level VHDL models can be used to evaluate and compare designs, but require significant effort to generate.

This paper presents a tool that was developed to evaluate the tradeoffs involved in using truncated multipliers in FIR filters. The tool is based on a package of Java classes that models the building blocks of computational systems, such as adders and multipliers. These classes generate VHDL descriptions, and are used by other classes in hierarchical fashion to generate VHDL descriptions of more complex systems. This paper describes the generation of truncated FIR filters as an example.

Previous techniques for modeling and designing digital signal processing systems with VHDL are presented in [1–5]. The tool described in this paper differs from those techniques by leveraging the benefits of object oriented programming (OOP). By subclassing existing objects, such as multipliers, the tool is easily extended to generate VHDL models that incorporate the latest optimizations and techniques.

Sections 1.1 and 1.2 provide background necessary for understanding the two's complement truncated multipliers used in the FIR filter architecture, which is described in Section 2. Section 3 describes the tool for automatically generating VHDL models of those filters. Synthesis results of specific filter implementations are presented in Section 4, with concluding remarks given in Section 5.

1.1 Two's Complement Multipliers

Parallel tree multipliers form a matrix of partial product bits, which are then added to produce a product. Consider an m-bit multiplicand, A, and an n-bit multiplier, B. If A and B are integers in two's complement form, then

$$A = -a_{m-1}2^{m-1} + \sum_{i=0}^{m-2} a_i 2^i \qquad \text{and} \qquad B = -b_{n-1}2^{n-1} + \sum_{j=0}^{n-2} b_j 2^j \quad . \tag{1}$$

Multiplying A and B together yields the following expression:

$$\begin{aligned} A \cdot B = a_{m-1}b_{n-1}2^{m+n-2} &+ \sum_{i=0}^{m-2}\sum_{j=0}^{n-2} a_i b_j 2^{i+j} \\ &- \sum_{i=0}^{m-2} b_{n-1}a_i 2^{i+n-1} - \sum_{j=0}^{n-2} a_{m-1}b_j 2^{j+m-1} \quad . \end{aligned} \tag{2}$$

The first two terms in (2) are positive. The third term is either zero (if $b_{n-1} = 0$) or negative with a magnitude of $\sum_{i=0}^{m-2} a_i 2^{i+n-1}$ (if $b_{n-1} = 1$). Similarly, the fourth term is either zero or a negative number. To produce the product of $A \times B$, the first two terms are added "as is". Since the third and fourth terms are negative (or zero), they are added by complementing each bit, adding '1' to the LSB column, and sign extending with a leading '1'. With these substitutions, the product is computed without any subtractions as:

$$\begin{aligned} P = a_{m-1}b_{n-1}2^{m+n-2} &+ \sum_{i=0}^{m-2}\sum_{j=0}^{n-2} a_i b_j 2^{i+j} + \sum_{i=0}^{m-2} \overline{b_{n-1}a_i} 2^{i+n-1} \\ &+ \sum_{j=0}^{n-2} \overline{a_{m-1}b_j} 2^{j+m-1} + 2^{m+n-1} + 2^{n-1} + 2^{m-1} \quad . \end{aligned} \tag{3}$$

Figure 1 shows the multiplication of two 8-bit integers in two's complement form. The partial product bit matrix is described by (3), and is implemented using an array of AND and NAND gates. The matrix is then reduced using techniques such as Wallace [6], Dadda [7], or Reduced Area reduction [8].

1.2 Truncated Multipliers

Truncated $m \times n$ multipliers, which produce results less than $m + n$ bits long, are described in [9]. Benefits of truncated multipliers include reduced area, delay, and power consumption [10]. An overview of truncated multipliers, which

						A		a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
						$\times B$		b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
							1	$\overline{a_7b_0}$	a_6b_0	a_5b_0	a_4b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0
							$\overline{a_7b_1}$	a_6b_1	a_5b_1	a_4b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1	
						$\overline{a_7b_2}$	a_6b_2	a_5b_2	a_4b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2		
					$\overline{a_7b_3}$	a_6b_3	a_5b_3	a_4b_3	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
				$\overline{a_7b_4}$	a_6b_4	a_5b_4	a_4b_4	a_3b_4	a_2b_4	a_1b_4	a_0b_4				
			$\overline{a_7b_5}$	a_6b_5	a_5b_5	a_4b_5	a_3b_5	a_2b_5	a_1b_5	a_0b_5					
		$\overline{a_7b_6}$	a_6b_6	a_5b_6	a_4b_6	a_3b_6	a_2b_6	a_1b_6	a_0b_6						
1	a_7b_7	$\overline{a_6b_7}$	$\overline{a_5b_7}$	$\overline{a_4b_7}$	$\overline{a_3b_7}$	$\overline{a_2b_7}$	$\overline{a_1b_7}$	$\overline{a_0b_7}$							
p_{15}	p_{14}	p_{13}	p_{12}	p_{11}	p_{10}	p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Fig. 1. 8×8 partial product bit matrix (two's complement)

discusses several methods for correcting the error introduced due to unformed partial product bits, is given in [11]. The method used in this paper is constant correction, as described in [9].

Figure 2 shows an 8×8 truncated parallel multiplier with a correction constant added. The final result is l-bits long. We define k as the number of truncated columns that are formed, and r as the number of columns that are not formed. In this example, the five least significant columns of partial product bits are not formed ($l = 8$, $k = 3$, $r = 5$).

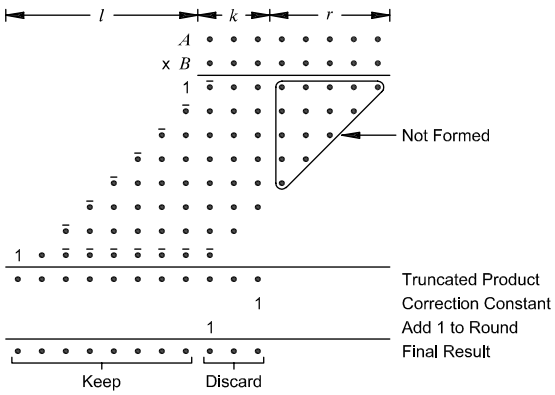

Fig. 2. 8×8 truncated multiplier with correction constant

Truncation saves an AND gate for each bit not formed and eliminates the full adders and half adders that would otherwise be required to reduce them to two rows. The delay due to reducing the partial product matrix is not improved because the height of the matrix is unchanged. However, a shorter carry propagate adder is required, which may improve the overall delay of the multiplier.

The correction constant, C_r, and the '1' added for rounding are normally included in the reduction matrix. In Figure 2 they are explicitly shown to make the concept more clear.

A consequence of truncation is that a reduction error is introduced due to the discarded bits. For simplicity, the operands are assumed to be integers, but the technique can also be applied to fractional or mixed number systems. With r unformed columns, the reduction error is

$$E_r = -\sum_{i=0}^{r-1}\sum_{j=0}^{i} a_{i-j}b_j 2^i \ . \tag{4}$$

If A and B are random with a uniform probability density, then the average value of each partial product bit is $\frac{1}{4}$, so the average reduction error is

$$E_{r\_avg} = -\frac{1}{4}\sum_{q=0}^{r-1}(q+1)2^q = -\frac{1}{4}((r-1)\cdot 2^r + 1) \ . \tag{5}$$

The correction constant, C_r, is chosen to offset E_{r_avg}. After rounding,

$$C_r = -\text{round}(2^{-r}E_{r\_avg})\cdot 2^r = \text{round}\left((r-1)\cdot 2^{-2} + 2^{-(r+2)}\right)\cdot 2^r \ , \tag{6}$$

where round(x) indicates x is rounded to the nearest integer.

2 FIR Filter Architecture

This section describes the architecture used to study the effect of truncated multipliers in FIR filters. Little work has been published in this area, and this architecture incorporates the novel approach of combining all constants for two's complement multiplication and correction of reduction error into a single constant added just prior to computing the final filter output. This technique reduces the average reduction error of the filter by several orders of magnitude, when compared to the approach of including the constants directly in the multipliers. Section 2.1 presents an overview of the architecture, and Section 2.2 describes components within the architecture.

2.1 Architecture Overview

An FIR filter with T taps computes the following difference equation [12],

$$y[n] = \sum_{k=0}^{T-1} b[k]\cdot x[n-k] \ , \tag{7}$$

where $x[\,]$ is the input data stream, $b[k]$ is the k^{th} tap coefficient, and $y[\,]$ is the output data stream of the filter. Since the tap coefficients and the impulse response, $h[n]$, are related by

$$h[n] = \begin{cases} b[n], & n = 0, 1, \ldots, T-1 \\ 0, & \text{otherwise,} \end{cases} \tag{8}$$

Equation (7) can be recognized as the discrete convolution of the input stream with the impulse response [12].

Figure 3 shows the block diagram of the FIR filter architecture used in this paper. This architecture has two data inputs, `x_in` and `coeff`, and one data output, `y_out`. There are two control inputs which are not shown, `clk` and `loadtap`.

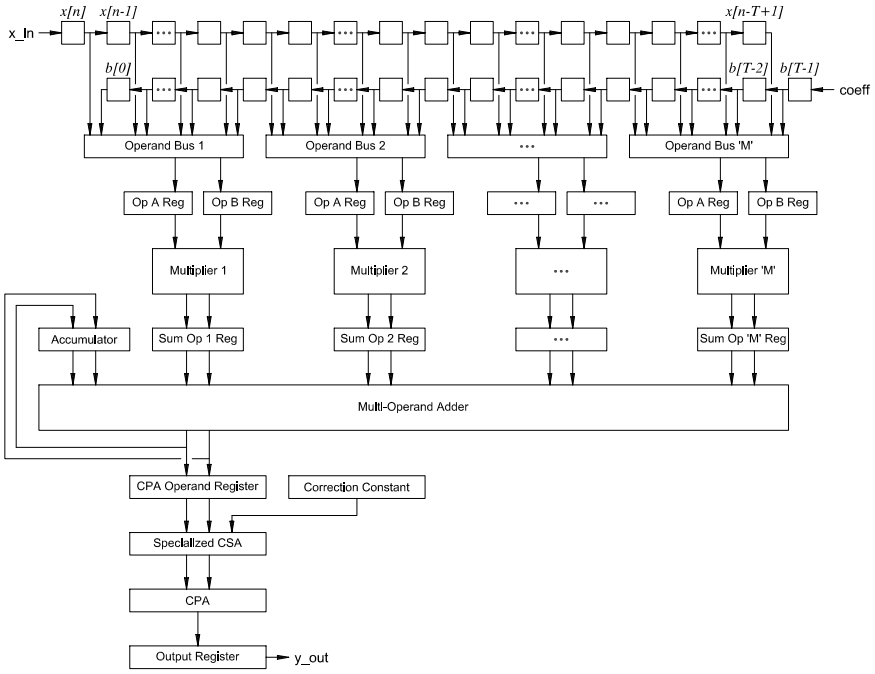

Fig. 3. Proposed FIR filter architecture with T taps and M multipliers

The input data stream enters at the `x_in` port. When the filter is ready to process a new sample, the data at `x_in` is clocked into the register labeled $x[n]$ in the block diagram. The $x[n]$ register is one of T shift registers, where T is the number of taps in the filter. When `x_in` is clocked into the $x[n]$ register, the values in the other registers are shifted right in the diagram, with the oldest value, $x[n-T+1]$ being discarded.

The tap coefficients are stored in another set of shift registers, labeled $b[0]$ through $b[T-1]$ in Figure 3. Coefficients are loaded into the registers by applying the coefficient values to the `coeff` port in sequence and cycling the `loadtap` signal to load each one.

The filter is pipelined with four stages: operand selection, multiplication, summation, and final addition.

Operand Selection: The number of multipliers in the architecture is configurable. For a filter with T taps and M multipliers, each multiplier performs

$\lceil T/M \rceil$ multiplications per input sample. The operands for each multiplier are selected each clock cycle by an operand bus and clocked into registers.

Multiplication: Each multiplier has two input operand registers, loaded by an operand bus in the previous stage. Each pair of operands is multiplied, and the final two rows of the reduction tree (the product in carry-save form) are clocked into a register where they become inputs to the multi-operand adder in the next stage. Keeping the result in carry-save form, rather than using a carry propagate adder (CPA), reduces the overall delay.

Summation: The multi-operand adder has carry-save inputs from each multiplier, as well as a carry-save input from the accumulator. After each of the $\lceil T/M \rceil$ multiplications have been performed, the output of the multi-operand adder (in carry-save form) is clocked into the CPA operand register where it is added in the next pipeline stage.

Final Addition: In the final stage, the carry-save vectors from the multi-operand adder and a correction constant are added by a specialized carry save adder and a carry propagate adder to produce a single result vector. The result is then clocked into an output register, which is connected to the `y_out` output port of the filter.

The `clk` signal clocks the system. The clock period is set so that the multipliers and the multi-operand adder can complete their operation within one clock cycle. Therefore, $\lceil T/M \rceil$ clock cycles are required to process each input sample. The final addition stage only needs to operate once per input sample, so it has $\lceil T/M \rceil$ clock cycles to complete its calculation and is generally not on the critical path.

2.2 Architecture Components

This section discusses the components of the FIR filter architecture.

Multipliers. In this paper, two's complement parallel tree multipliers are used to multiply the input data by the filter coefficients. When performing truncated multiplication, the constant correction method [9] is used. The output of each multiplier is the final two rows remaining after reduction of the partial product bits, which is the product in carry-save form [13]. Rounding does not occur at the multipliers, each product is $(l + k)$-bits long. Including the extra k bits in the summation avoids an accumulation of roundoff errors. Rounding is done in the final addition stage.

As described in Section 1.1, the last three terms in (3) are constants. In this architecture, these constants are *not* included in the partial product matrix. Likewise, if using truncated multipliers, the correction constant is not included either. Instead, the constants for each multiplication are added in a single operation in the final addition stage of the filter. This is described later in more detail.

Multi-operand Adder and Accumulator. As shown in (7), the output of an FIR filter is a sum of products. In this architecture, M products are computed per clock cycle. In each clock cycle, the carry-save outputs of each multiplier are added and stored in the accumulator register, also in carry-save form. The accumulator is included in the sum, except with the first group of products for a new input sample. This is accomplished by clearing the accumulator when the first group of products arrives at the input to the multi-operand adder.

The multi-operand adder is simply a counter reduction tree, similar to a counter reduction tree for a multiplier, except that it begins with operand bits from each input instead of a partial product bit matrix. The output of the multi-operand adder is the final two rows of bits remaining after reduction, which is the sum in carry-save form. This output is clocked into the accumulator register every clock cycle, and clocked into the CPA Operand Register every $\lceil T/M \rceil$ cycles.

Correction Constant Adder. As stated previously, the constants required for two's complement multipliers and the correction constant for unformed bits in truncated multipliers are not included in the reduction tree but are added during the final addition stage. A '1' for rounding the filter output is also added in this stage. All of these constants for each multiplier are precomputed and added as a single constant, C_{TOTAL}.

All multipliers used in this paper operate on two's complement operands. From (3), the constant which must be added for an $m \times n$ multiplier is $2^{m+n-1} + 2^{n-1} + 2^{m-1}$. With T taps, there are T multiply operations (assuming T is evenly divisible by M), so a value of

$$C_M = T(2^{m+n-1} + 2^{n-1} + 2^{m-1}) \tag{9}$$

must be added in the final addition stage.

The multipliers may be truncated with unformed columns of partial product bits. If there are unformed bits, the total average reduction error of the filter is $T \cdot E_{r\_avg}$. The correction for this is

$$C_R = \text{round}\left(T \cdot (r-1) \cdot 2^{-2} + T \cdot 2^{-(r+2)}\right) \cdot 2^r \ . \tag{10}$$

To round the filter output to l bits, the rounding constant that must be used is

$$C_{RND} = 2^{r+k-1} \ . \tag{11}$$

Combining these constants, the total correction constant for the filter is

$$C_{TOTAL} = C_M + C_R + C_{RND} \ . \tag{12}$$

Adding C_{TOTAL} to the multi-operand adder output is done using a specialized carry-save adder (SCSA) which is simply a carry-save adder optimized for adding a constant bit vector. A carry-save adder uses full adders to reduce three

bit vectors to two. SCSA's differ in that half adders are used in columns where the constant is a '0' and specialized half adders are used in columns where the constant is a '1'. A specialized half adder computes the sum and carry-out of two bits plus a '1', the logic equations being

$$s_i = \overline{a_i \oplus b_i} \qquad \text{and} \qquad c_{i+1} = a_i + b_i \ . \tag{13}$$

The output of the SCSA is then input to the final carry propagate adder.

Final Carry Propagate Adder. The output of the specialized carry-save adder is the filter output in carry-save form. A final carry propagate adder (CPA) is required to compute the final result. The final addition stage has $\lceil T/M \rceil$ clock cycles to complete, so for many applications a simple ripple-carry adder will be fast enough. If additional performance is required, a carry-lookahead adder may be used. Using a faster CPA does not increase throughput, but does improve latency.

Control. A filter with T taps and M multipliers requires $\lceil T/M \rceil$ clock cycles to process each input sample. The control circuit is a state machine with $\lceil T/M \rceil$ states, implemented using a modulo-$\lceil T/M \rceil$ counter. The present state is the output of the counter and is used to control which operands are selected by each operand bus. In addition to the present state, the control circuit generates four other signals: 1) `shiftData`, which shifts the input samples, 2) `clearAccum`, which clears the accumulator, 3) `loadCpaReg`, which loads the multi-operand adder output into the CPA operand register, and 4) `loadOutput`, which loads the final sum into the output register.

3 Filter Generation Software (FGS)

The architecture described in Section 2 provides a great deal of flexibility in terms of operand size, the number of taps, and the type of multipliers used. This implies that the design space is quite large. In order to facilitate the development of a large number of specific implementations, a tool was designed that automatically generates synthesizable structural VHDL models given a set of parameters. The tool, which is named FGS, also generates test benches and files of test vectors to verify the filter models.

FGS is written in Java and consists of two main packages. The arithmetic package, discussed in Section 3.1, is suitable for general use and is the foundation of FGS. The fgs package, discussed in Section 3.2, is specifically for generating the filters described previously. It uses the arithmetic package to generate the necessary components.

3.1 The arithmetic Package

The arithmetic package includes classes for modeling and simulating digital components. The simplest components include D flip-flops, half adders, and full adders. Larger components such as ripple-carry adders and parallel multipliers use the smaller components as building blocks. These components in turn are used to model complex systems such as FIR filters.

Common Classes and Interfaces. Figure 4 shows the classes and interfaces which are used by arithmetic subpackages. The most significant of these are VHDLGenerator, Parameterized, and Simulator.

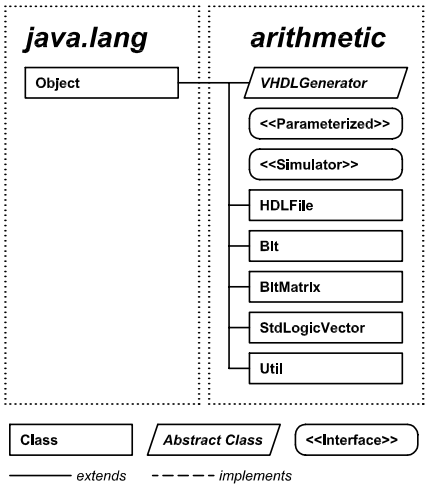

Fig. 4. The arithmetic package

VHDLGenerator is an abstract class. Any class that represents a digital component and can generate a VHDL model of itself is derived from this class. It defines three abstract methods which must be implemented by all subclasses. genCompleteVHDL() generates a complete VHDL file describing the component. This file includes synthesizable entity-architecture descriptions of all subcomponents used. genComponentDeclaration() generates the component declaration which must be included in the entity-architecture descriptions of other components which use this component. genEntityArchitecture() generates the entity-architecture description of this component.

Parameterized is an interface implemented by classes whose instances can be defined by a set of parameters. The interface includes get and set methods to access those parameters. Specific instances of Parameterized components can be easily modified by changing these parameters.

Simulator is an interface implemented by classes that can simulate their operation. The interface has only one method, simulate, which accepts a vector of inputs and returns a vector of outputs. These inputs and outputs are vectors of IEEE VHDL `std_logic_vectors` [14].

The arithmetic.smallcomponents Package. The arithmetic.smallcomponents package provides fundamental components including D flip-flops and full adders which are used as building blocks for larger components such as registers, adders, and multipliers. Each class in this package is derived from VHDLGenerator, enabling each to generate VHDL for use in larger components.

The arithmetic.adders Package. The classes in this package model various types of adders including carry propagate adders, specialized carry-save adders, and multi-operand adders. All components in these classes handle operands of arbitrary length and weight. This flexibility makes automatic VHDL generation more complex than it would be if operands were constrained to be the same length and weight. However, this flexibility is often required when an adder is used with another component such as a multiplier.

Figure 5 shows the arithmetic.adders package, which is typical of many of the arithmetic subpackages. CarryPropagateAdder is an abstract class from which carry propagate adders such as ripple-carry adders and carry-lookahead adders are derived. CarryPropagateAdder is a subclass of VHDLGenerator and implements the Simulator and Parameterized interfaces. Using interfaces and an inheritance hierarchy such as this help make FGS both straightforward to use and easy to extend. For example, a new type of carry propagate adder could be incorporated into existing complex models by subclassing CarryPropagateAdder.

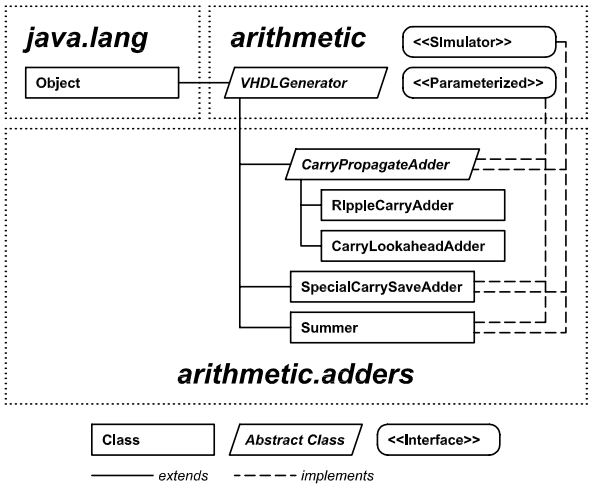

Fig. 5. The arithmetic.adders package

The arithmetic.matrixreduction Package. This package provides classes that perform matrix reduction, typically used by multi-operand adders and parallel multipliers. These classes perform Wallace, Dadda, and Reduced Area reduction [6–8]. Each of these classes are derived from the abstract class ReductionTree.

The arithmetic.multipliers Package. A ParallelMultiplier class was implemented for this paper and is representative of how FGS functions.

Parameters can be set to configure the multiplier for unsigned, two's complement, or combined operation. The number of unformed columns, if any, and the type of reduction, Wallace, Dadda, or Reduced Area, may also be specified. A BitMatrix object, which models the partial product matrix, is then instantiated and passed to a ReductionTree object for reduction. Through polymorphism (dynamic binding), the appropriate subclass of ReductionTree reduces the BitMatrix to two rows. These two rows can then be passed to a CarryPropagateAdder object for final addition, or in the case of the FIR filter architecture described in this paper, to a multi-operand adder.

The architecture of FGS makes it easy to change the bit matrix, reduction scheme, and final addition method. New techniques can be added seamlessly by subclassing appropriate abstract classes.

The arithmetic.misccomponents Package. This package includes classes that provide essential functionality but don't logically belong in other packages. This includes Bus, which models the operand busses of the FIR filter, and Register which models various types of data registers. Implementation of registers is done by changing the type of flip-flop objects which comprise the register.

The arithmetic.firfilters Package. This package includes classes for modeling ideal FIR filters as well as FIR filters based on the truncated architecture described in Section 2.

The "ideal" filters are ideal in the sense that the data and tap coefficients are double precision floating point. This is a reasonable approximation of infinite precision for most practical applications. The purpose of an ideal FIR filter object is to provide a baseline for comparison with practical FIR filters and allow measurement of calculation errors.

The FIRFilter class models FIR filters based on the architecture shown in Figure 3. All operands in FIRFilter objects are considered to be two's complement integers, and the multipliers and the multi-operand adder use Reduced Area reduction. There are many parameters that can be set including the tap coefficient and data lengths, the number of taps, the number of mulipliers, and the number of unformed columns in the multipliers.

The arithmetic.testing Package. This package provides classes for testing components generated by other classes, including parallel multipliers and FIR filters. The FIR filter test class generates a test bench and an input file of test vectors. It also generates a .vec file for simulation using Altera Max+Plus II.

The arithmetic.gui Package. This package provides graphical user interface (GUI) components for setting parameters and generating VHDL models for all of the larger components such as FIRFilter, ParallelMultiplier, etc. The GUI for each component is a Java Swing JPanel, which can be used in any Swing application. These panels make setting component parameters and generating VHDL files simple and convenient.

3.2 The fgs Package

Whereas the arithmetic package is suitable for general use, the fgs package is specific to the FIR filter architecture described in Section 2. fgs includes classes for automating much of the work done to analyze the use of truncated multipliers in FIR filters. For example, this package includes a driver class that automatically generates a large number of different FIR filter configurations for synthesis and testing. Complete VHDL models are then generated, as well as Tcl scripts to drive the synthesis tool. The Tcl script commands the synthesis program to write area and delay reports to disk files, which are are parsed by another class in the fgs package that summarizes the data and writes it to a CSV file for analysis by a spreadsheet application.

4 Results

Table 1 presents some representative synthesis results that were obtained from the Leonardo synthesis tool and the LCA300K 0.6 micron CMOS standard cell library. Additional data can be found in [15], which also also provides a more detailed analysis of the FIR filter architecture presented in this paper, including reduction and roundoff error. The main findings are:

1. Using truncated multipliers in FIR filters results in significant improvements in area. For example, the area of a 16-bit filter with 4 multipliers and 24 taps improves by 22.5 % with 12 unformed columns and by 36.4 % with 16 unformed columns. We estimate substantial power savings would be realized as well. Truncation has little impact on the overall delay of the filter.
2. The computational error introduced by truncation is tolerable for many applications. For example, the reduction error SNR for a 16-bit filter with 24 taps is 86.7 dB with 12 unformed columns and 61.2 dB with 16 unformed columns. In comparison, the roundoff error for an equivalent filter without truncation is 89.1 dB [15].
3. The average reduction error of a filter is independent of r (for $T > 4$), and much less than that of a single truncated multiplier. For a 16-bit filter with 24 taps and $r = 12$, the average reduction error is only 9.18×10^{-5} ulps, where an ulp is a unit of least precision in the 16-bit product. In comparison, the average reduction error of a single 16-bit multiplier with $r = 12$ is 1.56×10^{-2} ulps, and the average roundoff error of the same multiplier without truncation is 7.63×10^{-6} ulps.

Filter			Synthesis Results			Improvement			Reduction Error		
			Area	Total Delay	$A \cdot D$ Product		Total	$A \cdot D$	$\mathrm{SNR_R}$	σ_R	E_{AVG}
T	M	r	(gates)	(ns)	(gates·ns)	Area	Delay	Product	(dB)	(ulps)	(ulps)
12	2	0	16241	40.80	662633	—	—	—	∞	0	0
12	2	12	12437	40.68	505937	23.4%	0.3%	23.6%	89.70	0.268	-4.57E-5
12	2	16	10211	40.08	409257	37.1%	1.8%	38.2%	64.22	5.040	-4.57E-5
16	2	0	17369	54.40	944874	—	—	—	∞	0	0
16	2	12	13529	54.24	733813	22.1%	0.3%	22.3%	88.45	0.310	-6.10E-5
16	2	16	11303	53.44	604032	34.9%	1.8%	36.1%	62.97	5.820	-6.10E-5
20	2	0	19278	68.00	1310904	—	—	—	∞	0	0
20	2	12	15475	67.80	1049205	19.7%	0.3%	20.0%	87.48	0.346	-7.60E-5
20	2	16	13249	66.80	885033	31.3%	1.8%	32.5%	62.00	6.508	-7.60E-5
24	2	0	20828	81.60	1699565	—	—	—	∞	0	0
24	2	12	17007	81.36	1383690	18.3%	0.3%	18.6%	86.69	0.379	-9.18E-5
24	2	16	14781	80.16	1184845	29.0%	1.8%	30.3%	61.21	7.143	-9.18E-5
12	4	0	25355	20.40	517242	—	—	—	∞	0	0
12	4	12	18671	20.34	379768	26.4%	0.3%	26.6%	89.70	0.268	-4.57E-5
12	4	16	14521	20.04	291001	42.7%	1.8%	43.7%	64.22	5.040	-4.57E-5
16	4	0	26133	27.20	710818	—	—	—	∞	0	0
16	4	12	19413	27.12	526481	25.7%	0.3%	25.9%	88.45	0.310	-6.10E-5
16	4	16	15264	26.72	407854	41.6%	1.8%	42.6%	62.97	5.820	-6.10E-5
20	4	0	28468	34.00	967912	—	—	—	∞	0	0
20	4	12	21786	33.90	738545	23.5%	0.3%	23.7%	87.48	0.346	-7.60E-5
20	4	16	17636	33.40	589042	38.0%	1.8%	39.1%	62.00	6.508	-7.60E-5
24	4	0	29802	40.80	1215922	—	—	—	∞	0	0
24	4	12	23101	40.68	939749	22.5%	0.3%	22.7%	86.69	0.379	-9.18E-5
24	4	16	18950	40.08	759516	36.4%	1.8%	37.5%	61.21	7.143	-9.18E-5

Table 1. Synthesis results for 16-bit operands, output rounded to 16-bits (optimized for area)

5 Conclusions

This paper presents a tool used to rapidly prototype parameterized FIR filters. The tool is used to study the effects of using truncated multipliers in those filters. It is based on a package of arithmetic classes that are used as components in hierarchical designs, and are capable of generating structural level VHDL models of themselves. Using these classes as building blocks, FirFilter objects generate complete VHDL models of specific FIR filters. The arithmetic package is extendable and suitable for use in other applications, enabling rapid prototyping of other computational systems. As a part of ongoing research at Lehigh University, the tool is being expanded to study other DSP applications, and will be made available to the public in the near future.

References

1. Lightbody, G., Walke, R., Woods, R.F., McCanny, J.V.: Rapid System Prototyping of a Single Chip Adaptive Beamformer. (In: Proceedings of Signal Processing Systems) 285–294
2. McCanny, J., Ridge, D., Yi, H., Hunter, J.: Hierarchical VHDL Libraries for DSP ASIC Design. In: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing. (1997) 675–678
3. Pihl, J., Aas, E.J.: A Multiplier and Squarer Generator for High Performance DSP Applications. In: Proceedings of the 39th Midwest Symposium on Circuits and Systems. (1996) 109–112
4. Richards, M.A., Gradient, A.J., Frank, G.A.: Rapid Prototyping of Application Specific Signal Processors. Kluwer Academic Publishers (1997)
5. Saultz, J.E.: Rapid Prototyping of Application-Specific Signal Processors (RASSP) In-Progress Report. Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology (1997) 29–47
6. Wallace, C.S.: A Suggestion for a Fast Multiplier. IEEE Transactions on Electronic Computers **EC-13** (1964) 14–17
7. Dadda, L.: Some Schemes for Parallel Multipliers. Alta Frequenza **34** (1965) 349–356
8. Bickerstaff, K.C., Schulte, M.J., Swartzlander, Jr., E.E.: Parallel Reduced Area Multipliers. IEEE Journal of VLSI Signal Processing **9** (1995) 181–191
9. Schulte, M.J., Swartzlander, Jr., E.E.: Truncated Multiplication with Correction Constant. In: VLSI Signal Processing VI, Eindhoven, Netherlands, IEEE Press (1993) 388–396
10. Schulte, M.J., Stine, J.E., Jansen, J.G.: Reduced Power Dissipation Through Truncated Multiplication. In: IEEE Alessandro Volta Memorial Workshop on Low Power Design, Como, Italy (1999) 61–69
11. Swartzlander, Jr., E.E.: Truncated Multiplication with Approximate Rounding. In: Proceedings of the 33rd Asilomar Conference on Signals, Circuits, and Systems. (1999) 1480–1483
12. Oppenheim, A.V., Schafer, R.W.: Discrete-Time Signal Processing, 2nd edition. Prentice Hall, Upper Saddle River, NJ (1999)
13. Koren, I.: Computer Arithmetic and Algorithms. Prentice Hall, Englewood Cliffs, NJ (1993)

14. : IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std-logic1164): IEEE Std 1164-1993 (26 May 1993)
15. Walters III, E.G.: Design Tradeoffs Using Truncated Multipliers in FIR Filter Implementations. Master's thesis, Lehigh University (2002)