Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

VHDL Library of Arithmetic Units

Reto Zimmermann

Technical Report No. 98/3
January 1998

Abstract

A comprehensive library of arithmetic units written in synthesizable VHDL code has been developed. The library contains components for a variety of arithmetic operations and for different speed requirements. The library components are implemented as circuit generators in parameterized structural VHDL code. Their modular and well-documented source code allows for simple usage and easy customization. Highly efficient circuit architectures are used, which are optimized for synthesis and cell-based design. The VHDL library is platform independent, and it provides circuits with comparable performance, but higher flexibility and a larger diversity of arithmetic operations compared to commercial data path libraries.

This work was supported by MicroSwiss, a Microelectronics Program of the Swiss Government.

1 Introduction

Arithmetic units are universal and often performance-critical building blocks on ASICs. Commercial hardware synthesis tools and data path libraries provide efficient circuit generators for the most important arithmetic operations, starting from a behavioral circuit description e.g. in VHDL. However, the supplied arithmetic units usually lack flexibility for customization, and non-standard arithmetic functions are not supported.

To correct these deficiencies, a library of arithmetic units has been developed in this work. The requirements for such a library are:

- The library contains units for a comprehensive set of arithmetic operations.
- The units have multiple structurally different implementations with varying circuit performance, thus allowing the designer to trade area versus speed.
- The units make use of well-performing circuit architectures which are optimized for synthesis and cell-based design (standard cells, sea-of-gates, fine-grained FPGA)).
- The units are made available as circuit generators implemented in parameterized structural and synthesizable VHDL code. This provides technology (i.e. cell library) and platform (i.e. synthesis tool) independency as well as simple usage (i.e. by way of component instantiation in a VHDL circuit description).
- The units are implemented in a flexible and modular way in order to enable easy customization. This also calls for well-documented source code.

2 Foundations

2.1 Computer Arithmetic

The arithmetic basics, algorithms, and circuit structures used to implement the arithmetic units are described in detail in the accompanying lecture notes on "Computer Arithmetic: Principles, Architectures, and VLSI Design" [4] and in [5].

2.2 Cell-based VLSI

This work focuses on cell-based VLSI, which includes making use of logic cell libraries, circuit synthesis from HDL circuit descriptions, logic optimization, library binding, and automatic place and route.

The arithmetic circuit structures used are optimized so that they can easily be generated by simple algorithms. Redundancy removal – as part of the logic optimization – allows the generation redundant logic, which often makes circuit structure more regular and generators simpler.

The circuit structures are also optimized to take full advantage of typical cell libraries. For instance, the highly efficient full-adder cells are used wherever possible, and AOI/OAI-gates (AOI = AND-OR-INVERT) are preferred over multiplexers.

In particular, *standard-cell* technologies have been targeted. Similar results can be expected for *gate-array* or *sea-of-gates* ICs, which makes the library suitable for these technologies as well. *Fine-grained*

*FPGA*s (such as the Xilinx XC6200 family) have a logic cell granularity comparable to standard cells. Placement and routing is much more difficult due to the very limited routing resources. Therefore, it is an open question whether the approach of circuit synthesis and automatic place and route of arithmetic units is suitable for fine-grained FPGAs, or whether vendor-specific soft- or hard-macros have to be used. In *coarse-grained FPGA*s, the use of macros for arithmetic units is mandatory in order to take full advantage of the complex logic blocks and built-in fast carry logic.

2.3 Circuit Generators in VHDL

Circuit generators can be implemented in software programs. Such a program generates circuit descriptions in e.g. structural VHDL code which then can be processed further in the target hardware synthesis tool. However, platform independency and ease-of-use are not guaranteed using this approach.

In this work, circuit generators are directly implemented in parameterized structural VHDL code [1]. This way, no additional software is required and the library components can directly be used by instantiation in the designer's VHDL code. The VHDL constructs used for writing circuit generators are:

Generics for parameterization of components (e.g. for operand word width).

Concurrent signal assignment for the basic logic operations on signals.

Process for the behavioral description of simple operations for which the synthesis tool already generates efficient circuits.

Component instantiation for hierarchical description of complex circuits.

For-generate statement for repeated generation of concurrent statements.

If-generate statement for conditional generation of concurrent statements.

The absence of global (or control) variables in VHDL'87 (*shared variables* are only introduced in VHDL'93) makes the description of complex and irregular structures difficult or impossible. However, it turned out that all arithmetic circuits realized in this library can be described using regular structures which can be generated by the constructs listed above.

3 Implementation

3.1 VHDL Library

The VHDL library consists of a comprehensive set of arithmetic units and their subcomponents. For each component, a file exists containing their entity declaration and architecture body. All arithmetic components have two architecture bodies, one with a behavioral (for simulation) and the other with a structural (for synthesis) circuit description. An additional file provides the `arith_lib` package which contains all component declarations.

The usage of the library is straightforward, since its components can be instantiated in the VHDL source code in the same way as other circuit components.

3.2 Arithmetic Units

The implemented arithmetic units are summarized in Tables 1 and 2. Most units can be used both for unsigned and signed numbers. Some units have distinct implementations for unsigned and signed numbers.

Most units have multiple implementations with varying performance (can be chosen by a generic):

Slow implementations typically use linear circuit structures for critical operations, resulting in linear circuit size and linear circuit delay with respect to the operand width.

Fast implementations use tree structures for speeding up critical operations, resulting in circuit delays proportional to $\log n$ and circuit sizes proportional to n or $n \log n$ (n = operand width).

Medium speed implementations represent some compromise between fast and slow implementations. They typically make use of some smaller but slower tree arrangements.

3.3 Circuit Architectures

The following circuit architectures and optimization techniques are applied:

Tree structures instead of linear structures are used in reduction operations (e.g. Reduce-AND = AND concatenation of all bits in a vector), which in turn are used in some detectors. Tree arrangements of full-adders are used in counters and compressors. Tree structures basically have the same size as linear ones.

Prefix structures are used to solve prefix problems [4, 5]. Serial-prefix structures yield circuits with linear area and delay figures, while parallel-prefix circuits have a delay proportional to $\log n$ and area proportional to $n \log n$. One serial-prefix and two different parallel-prefix structures are used in all units containing Prefix-AND (detectors, incrementers), Prefix-AND-OR (adders, subtractors), and Prefix-XOR (Gray-to-binary converter) problems.

Wallace trees are used to speed up carry-save addition in multi-operand adders and multipliers.

Carry-save number representation of input operands and output results allows fast implementation of adders and multipliers without carry-propagate addition.

No Booth encoding was used in multipliers since this technique has not shown any performance advantages in standard-cell libraries with efficient full-adder cells.

3.4 Customization

The library components are described hierarchically in well-documented source code, which allows easy customization – like pipelining, internal word width adaptions, arithmetic optimizations, and combination of functions. Examples for customization are:

- A pipeline register can be placed between the carry-save adder and the final adder in all multipliers.

Table 1: Implemented arithmetic units.

Name	Description	Represent.		Performance		
		uns.	sgn.	slow	med.	fast
Adders						
Add	Adder	√	√	√	√	√
AddC	Adder with carry-in, carry-out	√		√	√	√
AddCfast	Adder with fast carry-in, carry-out	√		√	√	√
AddV	Adder with carry-in, 2's compl. overflow flag		√	√	√	√
AddMod2Nm1	Adder modulo $2^n - 1$ (double zero representation)	√		√	√	√
AddMod2Nm1s0	Adder modulo $2^n - 1$ (single zero representation)	√		√	√	√
AddMod2Np1	Adder modulo $2^n + 1$	√		√	√	√
AddCsv	Carry-save adder (3 operands)	√	√			√
AddMop	Multi-operand adder	√	√	√	√	√
AddMopCsv	Carry-save multi-operand adder	√	√	√		√
Subtractors, Complementers						
Sub	Subtractor	√	√	√	√	√
SubC	Subtractor with carry-in, carry-out	√		√	√	√
SubCZ	Subtractor with carry-in, carry-out, zero flag	√		√	√	√
SubV	Subtractor with carry-in, 2's compl. overflow flag		√	√	√	√
SubVZ	Subtractor with carry-in, 2's compl. ovl. & zero flag		√	√	√	√
Neg	2's complementer (negation)		√	√	√	√
NegC	2's complementer, conditional		√	√	√	√
AbsVal	Absolute value for 2's complement numbers		√	√	√	√
Adder-Subtractors						
AddSub	Adder-subtractor	√	√	√	√	√
AddSubC	Adder-subtractor with carry-in, carry-out	√		√	√	√
AddSubV	Adder-subtractor with carry-in, 2's compl. ovl. flag		√	√	√	√
Incrementers, Decrementers						
Inc	Incrementer	√	√	√	√	√
IncC	Incrementer with carry-in, carry-out	√		√	√	√
Dec	Decrementer	√	√	√	√	√
DecC	Decrementer with carry-in, carry-out	√		√	√	√
IncDec	Incrementer-decrementer	√	√	√	√	√
IncDecC	Incrementer-decrementer with carry-in, carry-out	√		√	√	√
Comparators						
CmpEQ	Equality comparator	√	√			√
CmpGE	Magnitude comparator	√		√	√	√
CmpEQGE	Equality and magnitude comparator	√		√	√	√
Multipliers						
MulSgn	Signed multiplier		√	√	√	√
MulUns	Unsigned multiplier	√		√	√	√
MulAddSgn	Signed multiplier-adder		√	√	√	√
MulAddUns	Unsigned multiplier-adder	√		√	√	√
AddMulSgn	Signed adder-multiplier		√	√	√	√
AddMulUns	Unsigned adder-multiplier	√		√	√	√
MulCsvSgn	Signed carry-save multiplier		√	√		√
MulCsvUns	Unsigned carry-save multiplier	√		√		√
SqrSgn	Signed squarer		√	√	√	√
SqrUns	Unsigned squarer	√		√	√	√

Table 2: Implemented arithmetic units (continued).

Name	Description	Represent. uns.	Represent. sgn.	Performance slow	Performance med.	Performance fast
Dividers, Square-Root Extractors						
DivArrSgn	Signed array divider		$\surd$	$\surd$		
DivArrUns	Unsigned array divider	$\surd$		$\surd$		
SqrtArrUns	Unsigned array square-root extractor	$\surd$		$\surd$		
Detectors						
AllZeroDet	All-zeroes detector					$\surd$
AllOneDet	All-ones detector					$\surd$
SumZeroDet	Fast zero-sum detection	$\surd$	$\surd$			$\surd$
LeadZeroDet	Leading-zeroes detector (LZD)	$\surd$		$\surd$	$\surd$	$\surd$
LeadOneDet	Leading-ones detector (LOD)	$\surd$		$\surd$	$\surd$	$\surd$
LeadSignDet	Leading-signs detector (LSD)		$\surd$	$\surd$	$\surd$	$\surd$
Log2	Integer logarithm (base 2)	$\surd$		$\surd$	$\surd$	$\surd$
Encoders, Decoders, Gray						
Decode	Decoder					$\surd$
Encode	Encoder					$\surd$
Bin2Gray	Binary-to-Gray converter					$\surd$
Gray2Bin	Gray-to-binary converter			$\surd$	$\surd$	$\surd$
IncGray	Gray incrementer			$\surd$	$\surd$	$\surd$
IncGrayC	Gray incrementer with carry-in			$\surd$	$\surd$	$\surd$
Miscellaneous						
Cnt	(m,k)-counter			$\surd$		$\surd$
Cpr	(m,2)-compressor			$\surd$		$\surd$
RedAnd	Reduce-AND					$\surd$
RedOr	Reduce-OR					$\surd$
RedXor	Reduce-XOR					$\surd$
PrefixAnd	Prefix-AND			$\surd$	$\surd$	$\surd$
PrefixOr	Prefix-OR			$\surd$	$\surd$	$\surd$
PrefixAndOr	Prefix-AND-OR			$\surd$	$\surd$	$\surd$
PrefixXor	Prefix-XOR			$\surd$	$\surd$	$\surd$

- If only the upper half word of a multiplier result is used, the word width can already be reduced after the carry-save addition of the partial products, thus reducing the size of the carry-save adder and the final adder.
- Multiple additions and multiplications in series can be made faster using carry-save adders and multipliers.
- Arbitrary components or subcomponents can be combined in order to implement specialized operations.
- Additional output signals from a component are required.
- Functionality is changed or added to a component.

3.5 Versatile Adder Synthesis

A universal algorithm for the optimization and synthesis of parallel-prefix adders has been developed [3]. It bases on a graph optimization problem and generates area-optimized adders under arbitrary timing constraints including non-uniform signal arrival and required times. A Java Applet has been developed which implements the synthesis algorithm and a graphical user interface [2].

3.6 Verification

All library components have been funcionally verified by VHDL testbenches. Exhaustive simulations have been carried out for small word widths, comparing the results of behavioral and structural circuit descriptions. The regularity and parameterizability of the circuit structures used guarantees correct functionality also for larger word widths.

3.7 Documentation

The arithmetic units are documented in more detail in the VHDL source code files. The circuit architecture is shortly described in the file header. A behavioral description is given in the corresponding "Behavioral" architecture.

3.8 Performance Comparisons

Performance comparisons between some arithmetic units from the VHDL library and the corresponding units from the data path synthesizer by COMPASS Design Automation have been carried out. All circuits were synthesized and optimized using the COMPASS ASIC Synthesizer v4.2 for a 0.35 μm standard-cell library under typical PTV conditions. Table 3 gives the results.

Table 3: Performance comparisons.

Unit	Performance	COMPASS		VHDL library	
		# gates	delay (ns)	# gates	delay (ns)
Adder (16-bit)	slow	86	8.61	86	8.61
	medium	217	3.58	263	2.59
	fast	475	2.22	309	2.15
Multiplier (16-bit, unsigned)	slow	1884	23.92	1615	23.32
	medium	2422	12.02	2120	9.96
	fast	3725	9.02	2245	9.26

3.9 Software Platforms

The library has been tested exhaustively (i.e. all components) using the VHDL synthesis tool by COMPASS Design Automation for synthesis and the V-System VHDL simulator by Model Technology for simulation. The library has also been marginally tested (i.e. only selected components) on other software platforms. Table 4 gives an overview of the platforms considered.

Table 4: Software platforms.

Name	Vendor	Simulation	Synthesis	Exhaustive
ASIC Synthesizer	COMPASS Design Automation		√	√
V-System	Model Technology	√		√
QuickHDL	Mentor Graphics	√		
Autologic II	Mentor Graphics		√	
Leapfrog	Cadence Design Systems	√		
Synergy	Cadence Design Systems		√	
VHDL System Simulator	Synopsys	√		
Design Compiler	Synopsys		√	

4 Conclusions

This work shows that it is possible to write circuit generators for high-performance cell-based arithmetic units directly in parameterized structural VHDL code. On that basis, a comprehensive library of flexible components has been developed which allow simple and portable usage and easy customization. The library is made available on the World Wide Web [2].

References

[1] H. Kunz and R. Zimmermann. *High-Performance Adder Circuit Generators in Parameterized Structural VHDL.* Technical Report No. 96/7, Integrated Systems Laboratory, ETH Zürich, August 1996.

[2] R. Zimmermann. *VHDL Library of Arithmetic Units.* `<http://www.iis.ee.ethz.ch/~zimmi/arith_lib/>`.

[3] R. Zimmermann. Non-heuristic optimization and synthesis of parallel-prefix adders. In *Proc. Int. Workshop on Logic and Architecture Synthesis*, pages 123–132, Grenoble, France, December 1996.

[4] R. Zimmermann. *Computer Arithmetic: Principles, Architectures, and VLSI Design.* Lecture notes, Integrated Systems Laboratoty, ETH Zürich, 1997.

[5] R. Zimmermann. *Binary Adder Architectures for Cell-Based VLSI and their Synthesis.* PhD thesis, Swiss Federal Institute of Technology (ETH) Zurich, Hartung-Gorre Verlag, 1998.