

0.13 μm CMOS Synthesis of Common Arithmetic Units

Anders Lindström, Michael Nordseth and Lars Bengtsson

Technical Report 03-11, August 2003.

Department of Computer Engineering
Chalmers University of Technology

Abstract - This report presents a design space mapping of common arithmetic units with a 0.13 μm CMOS process technology. A behavioural vs. structural (gate-level logic) code comparison is made on key arithmetic functions, binary modulo arithmetic is studied and a brief complexity evaluation is made.

Keywords - VHDL, binary arithmetic, synthesis, behavioural code, structural code.

1 Introduction

Comprehensive synthesis data (delay, area and power performance) for common arithmetic circuits is seldom readily available, especially for newer process technologies such as 0.13 μm . This lack of reference data often results in poor design choices and sometimes also in needless work with custom solutions where standard circuits would meet the design specifications.

In this work we present a database of synthesis results for common arithmetic circuits, which origin from a public domain VHDL library (Arith Lib) written by Dr. Reto Zimmermann at Swiss Federal Institute of Technology [1]. The library contains units for a variety of arithmetic operations and for different speed requirements. The circuits were synthesized for all available speed options and for a set of different word lengths. A reference synthesis were also carried out, in which the synthesis tool was setup to interpret a behavioural circuit description under certain timing/area constraints. This enabled us to compare the performance of the circuits in Arith Lib to the standard designs predefined in, and used by, the synthesis tool.

2 Foundations

2.1 VHDL Library of Arithmetic Units

The arithmetic units in Arith Lib are word length generic and have multiple architectures suitable for different speed constraints. This flexibility is achieved by circuit generators that are directly implemented as parameterized structural VHDL code.

Units for several common arithmetic operations are provided, such as addition and multiplication, but also for operations such as modulo addition.

2.2 Synthesis Tools - Constraints and Process Technology

The synthesis tool used was Synopsys Design Compiler. The designs in Arith Lib are compiled without any constraints as timing constraints in this case are controlled by a generic variable. This generic variable selects one of three different architectures in Arith Lib corresponding to different timing constraints, slow (speed 0), medium (speed 1) and fast (speed 2) circuit implementations.

In the case of the reference synthesis, no constraints were given for the lowest speed setting (speed 0) and maximum timing constraints for the highest speed setting (speed 2). The circuits have been compiled under typical operating and wire load conditions in both cases.

The process technology used is UMC's 0.13 μm CMOS cell library with up to 8 layers of copper interconnects and a core voltage of 1.2V.

3 Behavioural code synthesis

This sections will focus on the differences in performance between gate-level logic code and behavioural code. The synthesis results for some representative units are presented and analysed.

3.1 Adders and subtractors

Figure 1 shows the performance of different adder structures from Arith Lib. The performance of the Arith Lib

implementations of Brent-Kung (speed 1) and Sklansky (speed 2) adder structures are nearly identical. One of the explanations of this might be that the synthesis tool have some difficulties interpreting the complex VHDL circuit generator code used in Arith Lib.

The adders in Arith Lib can be compared with the adder structures generated directly by the synthesis tool which are shown in Figure 2. The adder circuits generated by the synthesis tool are identical to those from Arith Lib in the speed 0 case (i.e. ripple-carry structure), but the differences are more substantial in the speed 2 case. This indicates that a greater design space is available when using a behavioural description of an adder and using constraint driven synthesis.

The adders in Arith Lib are, however, very suitable for customizations such as pipelining where it is impossible to use the default adder structures.

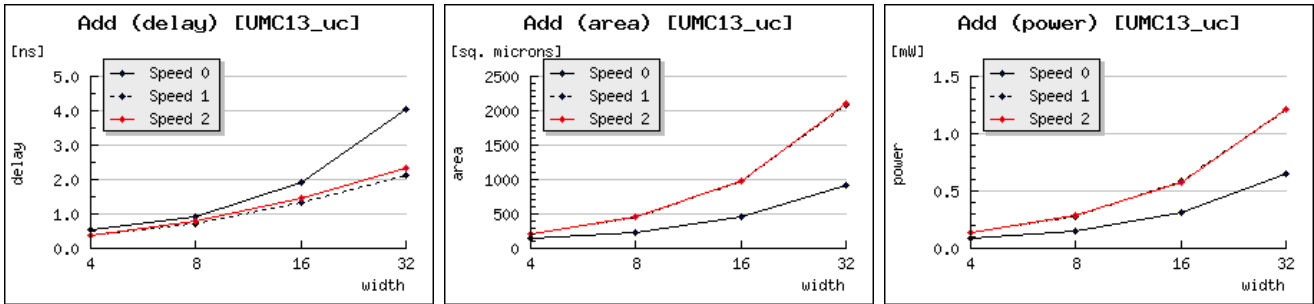


FIGURE 1 Ripple-carry (speed 0), Brent-Kung (speed 1) and Sklansky (speed 2) adders

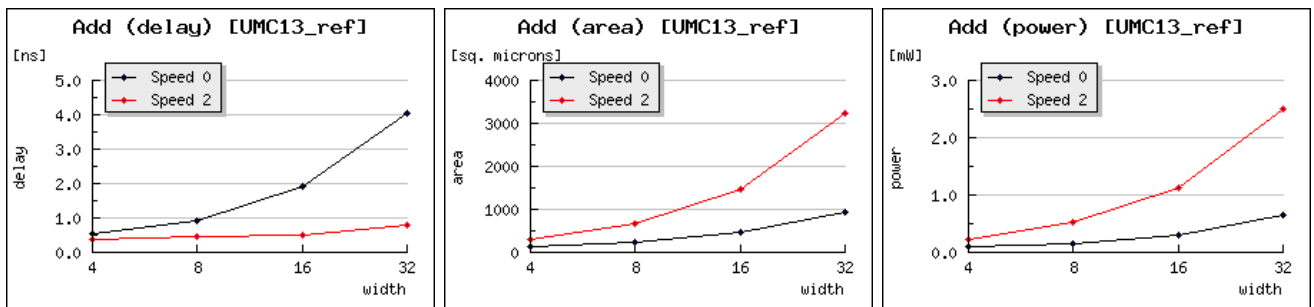


FIGURE 2 No constraints (speed 0) and hard timing constraints (speed 2) default adders

3.2 Multipliers

The performance of the signed multiplier in Arith Lib is shown in Figure 3. The difference between speed 1 and speed 2 is once again minimal. When compared to the reference synthesis shown in Figure 4, some interesting observations can be made; there is a significant difference in area, but the delay is almost identical for the highest speed setting. However, the speed 2 multiplier in Arith Lib has better power performance than the reference multiplier.

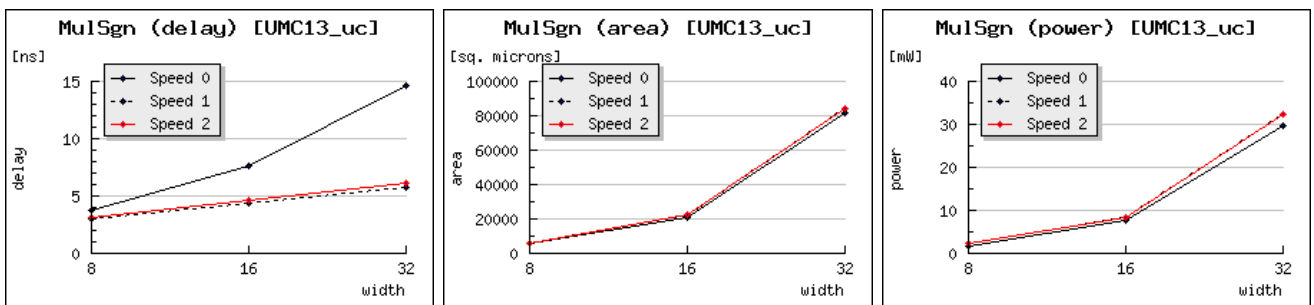


FIGURE 3 Signed multiplier

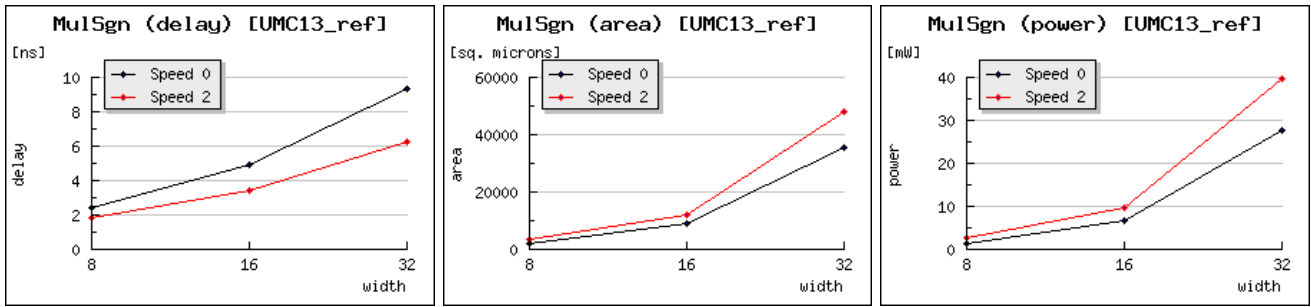


FIGURE 4 No constraints (speed 0) and hard timing constraints (speed 2) default multipliers

3.3 Miscellaneous

Figure 5 shows an example of one of the more nonstandard units in Arith Lib. This graycode to binary converter can also be implemented with a behavioural description which is shown in Figure 6. The behavioural description has overall worse performance this time.

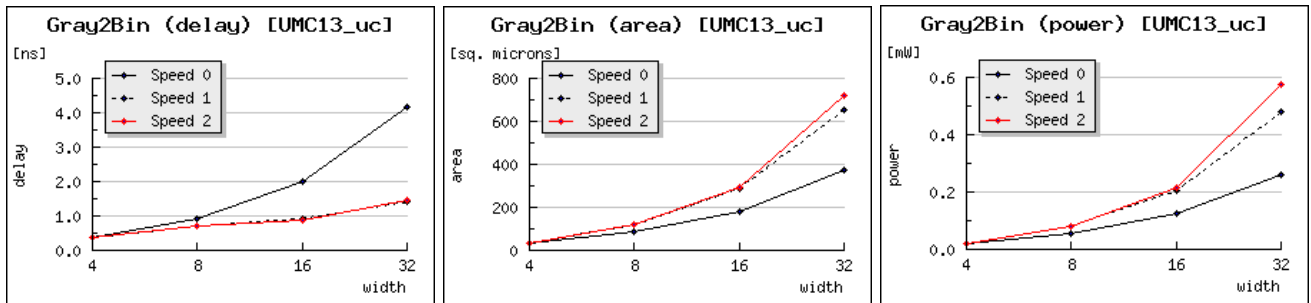


FIGURE 5 Graycode to binary converter

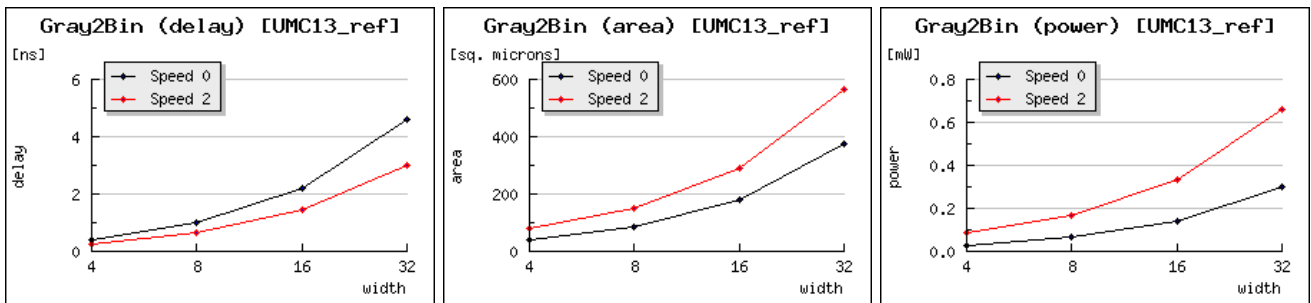


FIGURE 6 Graycode to binary converter with behavioral VHDL

4 Modulo arithmetic synthesis

Another interesting area is binary modulo arithmetic. Arith Lib contains some relatively new modulo adder designs [2]. Conventionally, binary modulo $2^n \pm 1$ adders are usually realized in one of the following ways:

- Binary adders with end-around-carry. This results in higher latency as the feedback carry increases the critical path.
- Binary addition in two steps, the first step is the add operation and the second handles the carry-out from the first.
- An binary adder followed by an incrementer.
- Two binary adders in parallel that calculates both possible results (i.e. one with a carry-in and one without).

All of these solutions results in larger and/or slower circuits than with a normal binary adder.

The modulo circuits in Arith Lib uses a single parallel-prefix structure that only requires one extra prefix level compared to parallel-prefix adders. However, there are some drawbacks with this solution; $2^n \pm 1$ addition must be

done with a diminished-one number representation or similar and will thus require conversion to and from this representation. Figure 7 and Figure 8 shows the performance of these modulo adders.

The 2^n-1 modulo adders and the 2^n+1 adders have similar performance and are only slightly worse than a normal adder (Figure 1). The drawback is that the modulo 2^n+1 adder uses diminished-one number representation and will thus require a binary adder to convert to and from this representation and the number zero can not be represented and must be handled and detected separately.

These drawbacks often limits the usefulness of binary modulo arithmetic to special applications such as cryptography.

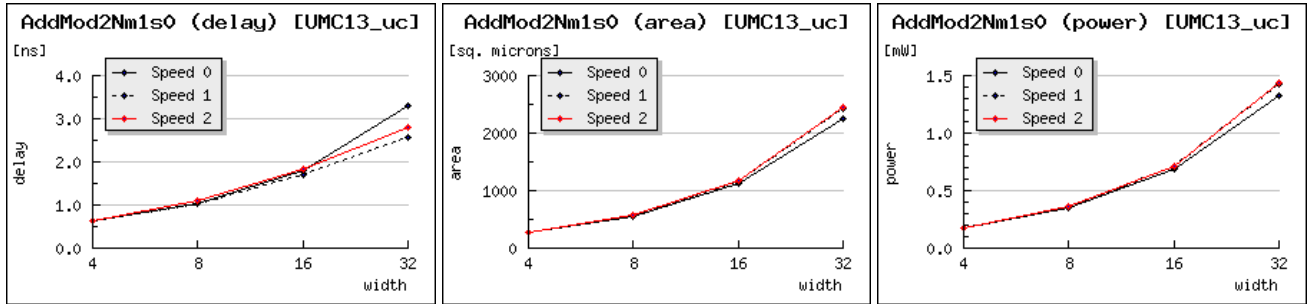


FIGURE 7 Modulo 2^n-1 adder

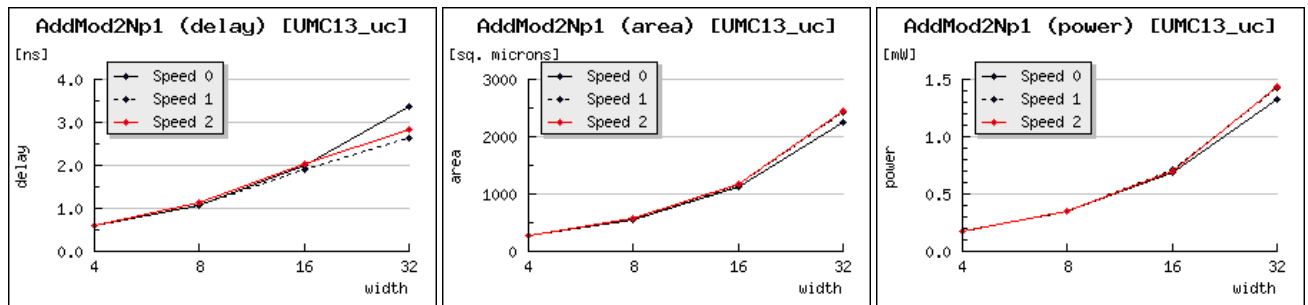


FIGURE 8 Modulo 2^n+1 adder

5 Complexity

The theoretical complexity for some of the arithmetic units in the library are well known [3]. But how well does it correlate with the complexity obtained in real synthesis? This section makes a comparison between the synthesis data in the database and the theoretical complexities found in literature.

5.1 Adders

Addition is the most common arithmetic operation and also serves as a building block for synthesizing other operations. The three common adder structures (previously mentioned in section 3.1) used in Arith Lib are ripple-carry adder (RCA), parallel-prefix Brent-Kung (PPA-BK) and parallel-prefix Sklansky (PPA-SK). Table I shows the theoretical complexities for those adders [3].

TABLE I Asymptotic adder complexities (unit-gate model)

adder type	area	delays	architecture
RCA	$7n$	$2n$	ripple-carry
PPA-BK	$10n$	$4 \log n$	parallel-prefix Brent-Kung
PPA-SK	$3/2 n \log n$	$2 \log n$	parallel-prefix Sklansky

The linearity of the RCA as well as the logarithmic behaviour of the PPAs is easy to recognize in the synthesis data, see Figure 9. However, the internal relationship between the PPAs is not the same as in the theoretical results, the Brent-Kung adders are marginally faster than the Skalnsky adders and both structures have the same area performance in our synthesis.

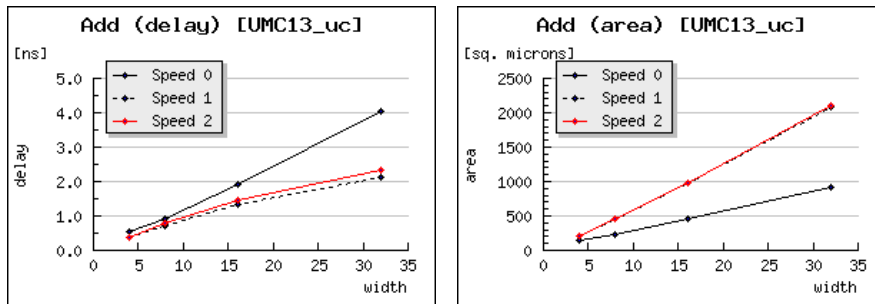


FIGURE 9 Adder with linear scales: RCA (speed 0), PPA-BK (speed 1) and PPA-SK (speed 2)

5.2 Multipliers

The multipliers in Lib have an array multiplier structure (Braun for unsigned and Baugh-Wooley for signed arithmetic) using a carry-save addition (CSA) tree which totals in the delay complexity $O(\log n + T_{\text{add}})$ and in the area complexity $O(n^2)$. Figure 10 shows the synthesis results of the Braun multiplier. The different speed settings only affects the type of CSA tree and the architecture of the final product adder, i.e. the delay complexity is $O(n)$ for speed 0 and $O(\log n)$ for speed 1 and speed 2.

The synthesis data seems to fit with the theoretical complexities (keeping in mind the few number of samples in this synthesis), but the difference between speed 1 and speed 2 are once again negligible.

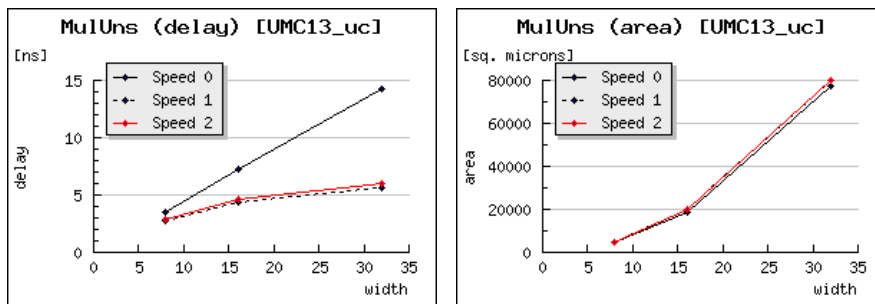


FIGURE 10 Unsigned multiplier with linear scales

6 Conclusions

A large number of arithmetic units have been synthesized and thus provided a design space mapping. This paper has presented some aspects of the evaluation of this mapping.

Behavioural code in VHDL can be used with good results if the arithmetic operation in question is common and have good support in the synthesis tool. Both addition and multiplication can thus be written in behavioural code to save time not implementing a gate-level logic description.

In the case of more uncommon operations, such as graycode decoding, some more thought must be given. In this case a gate-level logic description has better performance, but is it enough to spend time implementing it? When implementing arithmetic units that do not fit in any “standard-template” at all, the answer is much simpler as it is very hard to avoid gate-level logic in e.g. pipelining arithmetic operations.

It was shown that it is possible to implement efficient binary modulo arithmetic if the drawbacks of a diminished-one number representation can be accepted. This is, however, not the case in most applications where custom solutions or other number representations must be used.

The complete database of synthesis results can be found in [4].

References

- [1] R. Zimmermann. *VHDL Library of Arithmetic Units*. [Online]. Available: http://www.iis.ee.ethz.ch/~zimmi/arith_lib/
- [2] R. Zimmermann, "Efficient VLSI Implementation of Modulo ($2^n \pm 1$) Addition and Multiplication", in *IEEE Proc. 14th Symp. Computer Arithmetic*, pp. 158-167, April 1999.
- [3] R. Zimmermann, "*Binary Adder Architectures for Cell-Based VLSI and their Synthesis*", Diss. ETH No. 12480, 1997.
- [4] A. Lindström and M. Nordseth. (2003, Mars). *Arithmetic Database*. [Online]. Available: <http://www.ce.chalmers.se/arithdb/>