Layout Driven Optimization of Datapath Circuits using Arithmetic Reasoning

Ingmar Neumann* Dominik Stoffel* Kolja Sulimma* Michel Berkelaar+ Wolfgang Kunz*

*University of Kaiserslautern, Germany
Dept. of Electrical and Computer Engineering
Electronic Design Automation Group

+Magma Design Automation Inc.
2 Results Way
Cupertino, CA 95014, USA

Abstract

This paper proposes a new formalism for layout-driven optimization of datapaths. It is based on preserving an arithmetic bit level representation of the arithmetic circuit portions throughout various design stages. The arithmetic bit level description takes into account the arithmetic nature of the datapath and facilitates arithmetic reasoning to identify circuit transformations that are too complex to derive for Boolean reasoning. It is a bit-level representation so that it integrates well into standard design flows. Based on this representation we developed an optimization algorithm for cycle time. It takes interconnect delay into account and can be applied at late design stages. A prototype has been integrated into a commercial EDA environment. For circuits implementing complex arithmetic expressions we achieved performance improvements of up to 32%.

1. Introduction

This paper describes an optimization method that is targeted towards the adder tree structures that are common in datapath circuits. Datapath circuits often form a significant part of a modern integrated circuit (IC), and in many cases the critical timing path of these ICs passes through them. Traditional logic synthesis techniques [1, 5, 7, 8], which perform well on the control parts of the logic of the IC, are not well suited to optimize datapaths. Logic synthesis techniques based on algebraic optimizations are too limited to use the full spectrum of optimizations that are potentially available in datapaths due to the many existing symmetries. Sometimes they make the situation even worse, e.g., by destroying the regularity typical for arithmetic circuits[6]. Traditional techniques based on Boolean reasoning, like rewiring [10-13], could potentially exploit these symmetries. In practice, these methods are severely limited by the inherent computational intractability of finding all of the symmetries in datapath structures like multipliers. Finding all symmetries in a multiplier is computationally as hard as formally verifying the multiplier!

We propose a method that opens up the full use of optimization potential given by the symmetries in the datapaths with the computational efficiency of algebraic methods. To find the symmetries we propose to use an arithmetic bit level representation of the circuit. This representation models both arithmetic information and bit-level circuit structure. It has already been applied successfully in the context of formal equivalence checking [9]. The arithmetic bit level representation exposes all symmetry information we need, and as it does not change during optimization, we have an extremely fast way to use this global Boolean property to optimize datapaths. In this paper, we examine its use in layout-driven synthesis. We present an approach for optimizing an arithmetic circuit in terms of timing by swapping arithmetically symmetric operands. Our algorithm is also suited for being applied during layout independent logic synthesis as well as during combined synthesis/layout generation.

The remainder of this paper is organized as follows. Section 2 explains the arithmetic bit level structure. Section 3 shows how we can use it to find candidates for global operand swaps, and shows that the optimization potential opened up by this can change any adder tree structure into any other. Section 4 shows an application of the operand swaps in an algorithm performing timing optimization on a placed datapath circuit. Section 6 shows experimental results highlighting the tremendous optimization potential that is opened up: up to 32% of cycle time improvement at only 2% of area cost (all measured in routed results).

2. Arithmetic Bit Level

In the past, there was little reason to apply logic synthesis to arithmetic circuits. For implementing arithmetic expressions, module generators using constructive algorithms [2-4] are available. These approaches worked sufficiently well for synthesizing stand-alone arithmetic circuits, where all inputs and outputs have the same data arrival and required time, respectively. If an arithmetic expression is embedded into a large circuit the situation becomes more complicated. The module generator must take external constraints into account which may change substantially during the design process.

In recent years, an even more crucial difficulty has come up for module generators. The increasing contribution of interconnect delay to cycle time invalidates the simple timing models being used by most constructive algorithms. Without any layout data, predicting delay with any degree of accuracy is virtually impossible making pre-layout timing optimizations a game of chance.

To overcome these problems, in recent years a significant amount of effort has been spent in developing approaches for integrating logic optimization into the physical design flow. Post-placement optimization followed by placement modifications[8], initial placement of an unmapped netlist followed by logic synthesis and final placement[5], integrating logic optimization into the inner loop of a placement algorithm[7] are among the most promising. However, most of these methods are based on traditional approaches for logic optimization and, hence, they achieve adequate results for control logic only. [14] describes an optimization loop consisting of re-synthesizing adder trees constructively followed by repeating placement. The results being presented are promising. However, they still leave room for improvement due to the simple timing model being used. Another drawback is that several time consuming placement steps are required.

Arithmetic circuits have the remarkable property that arithmetic transformations may be used for optimizing them. This is done by high level datapath generators in practice. On the gate level, however, in general this is not possible since module generators produce gate netlists containing no information about the arithmetic behavior of the circuit.

In order to still enable arithmetic transformations on a gate-level description we propose the following approach: Additionally to the gate netlist the module generator creates a second description of the circuit (an arithmetic bit level description) preserving all the knowledge we need to perform arithmetic transformations at the bit level. This description is maintained throughout the design process allowing arithmetic optimizations during late design stages.

A circuit structure found in many kinds of datapaths are addition networks. They are the main component of multipliers as well as of complex arithmetic expressions as they are used in signal processing circuits. Internally, addition networks consist of cascaded addition trees. An addition tree computes the modulo-2 sum of n 1-bit operands. Furthermore, it computes $\lfloor n/2 \rfloor$ carry signals which are fed as input signals into the next addition tree. By cascading addition trees the number of carries produced in one addition tree is reduced until we finally obtain a $\lceil \log_2(n) \rceil$-bit wide operand consisting of $\lceil \log_2(n)\text{-}1 \rceil$ signals driven by sum outputs and one signal driven by a carry output. This operand represents the result of the addition. Typically, an addition tree is implemented using $(n\text{-}1)/2$ full adders if n is odd and $n/2\text{-}1$ full adders and one half adder if n is even. Fig. 1 shows addition trees consisting of three full adders and one full adder, respectively.

Remarkably all inputs to 1-bit adders belonging to the same addition tree are symmetric[15]. This opens up a large optimization potential. In the example shown in Fig. 1 swapping two signals reduces the delay of the critical path from 4 to 3 assuming unit delay for each full adder.

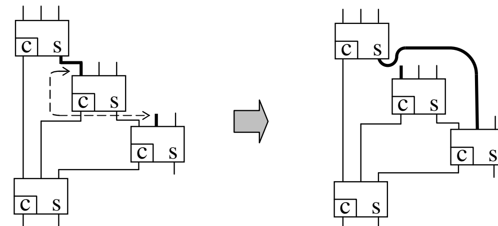

Fig. 1: Optimizing addition trees

Note that identifying such symmetries on the logic gate level by using conventional Boolean reasoning techniques is very expensive, often prohibitively so. However, if knowledge about the arithmetic nature of the circuit is preserved, only a simple analysis is required to identify all symmetries.

Therefore, we represent each addition network (*AN*) of a circuit as a set of cascaded addition trees (*AT*). Each *AT* is represented by a netlist consisting of 1-bit-addition units (*AU*). Each *AU* has either 2 or 3 inputs and calculates a sum and a carry output. An *arithmetic bit level description* of a circuit is a netlist where sub-circuits implementing addition networks are expressed exclusively in terms of *AU*s. All other parts of the circuit can be described using arbitrary gate types. This description allows us to identify groups of symmetric operands in linear time by traversing the *AN*s. For a formal definition of *AN*, *AT* and the *arithmetic bit level* we refer the reader to [15]. Our approach is based on the assumption that the arithmetic bit level representation is preserved by the synthesis tool through all design stages enabling arithmetic reasoning *beyond* the module generation phase, i.e., also during later design stages.

3. Operand swaps

Fig. 1 illustrated the optimization potential of exchanging symmetric operands. Syntactically, operand swapping looks like the well-known pin swapping technique that is standard in many design flows. Note however, that the operand swapping considered here performs transformations that are much more global than conventional pin swapping.

The powerfulness of this technique is described by the following theorem:

Theorem 1 : An addition tree of arbitrary topology can be transformed into <u>any</u> other topology by applying a sequence of operands swaps.

Proof: [15]■

Theorem 1 tells us that any embedded addition tree being built manually by a designer or by a constructive algorithm can be restructured to optimize timing by only performing operand swaps during the design flow.

Since addition networks are constructed by cascading addition trees this result obviously holds also for com-

plete addition networks. E.g. an array multiplier can be transformed into a Wallace tree architecture (or anything in between) simply by swapping operands. To better understand this keep in mind that the number of carries produced in one particular addition column during a multi-operand addition depends only on the number of 1-bit operands that have to be added up. Consequently, the number of carries being produced by the corresponding addition tree is the same for all possible addition networks implementing a particular multi-operand addition.

4. Timing Optimization

This section starts describing how to find promising operands swaps. Consider two input pins *A* and *B* of two different *AUs* as shown in Fig. 2. For each of them we can determine the delay t_{pd} of the longest path leading through it by adding the corresponding data arrival time t_{ar} and data time-to-sink t_{2s}. If we swap the signals *a* and *b* we simply have to swap the t_{ar} values in the corresponding equations and we have to add correcting factors C_{AB} and C_{BA} representing the changes in wire delays.

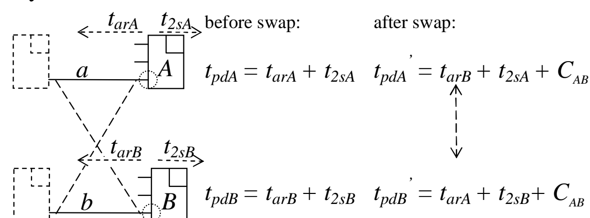

Fig. 2: Gain of an Operand swap

The maximum delay reduction achievable by the swap results to $gain = \max(t_{pdA}, t_{pdB}) - \max(t_{pdA}', t_{pdB}')$.

Since some path leading neither through pin *A* nor through pin *B* may become the most critical path after the swap the delay reduction may be smaller than *gain*. However, it can be guaranteed that performing a swap with positive gain will at least not deteriorate cycle time.

Note that the gain calculation is not bound to a certain timing model. Delay can be calculated with the accuracy of the data being available at a particular design level. This allows us to take all technology dependent parameters influencing delay like transistor sizes, wire capacitances, etc., into consideration during optimization. This is a fundamental advantage over the simple timing models being used in constructive algorithms.

Fig. 3 presents our algorithm *optimize*. It improves performance by swapping operands repeatedly and stops when no swap with positive gain is found on the critical path. Two signals are swap candidates for each other if they belong to the same *AT* and if they do not lie in the fanin cone of each other.

```
repeat {
    perform timing analysis;
    determine most critical path p_critical
    maxgain=0;
    Signal a,b;
    foreach Signal i ∈ p_critical being an input of an AU
        foreach Signal j being a swap candidate for i
            if (gain(i,j)>maxgain) {
                a=i; b=j; maxgain=gain(i,j);
    if (maxgain>0)
        swap signals a and b;
} until (maxgain=0);
```

Fig. 3: Algorithm *optimize*

Note that *optimize* does not consider particular addition trees in isolation. At any time it analyzes the critical path of the whole circuit. This path may lead through several addition trees (or even through several addition networks). The algorithm always globally searches for the most promising operand swap.

As explained in the previous sections this allows transforming an addition network for adding up several multi-bit operands into any other addition network implementing the same arithmetic function. Consequently, our method is orthogonal to specific multiplier architectures. Any architecture based on addition networks can be optimized using our approach.

The addition networks targeted by the algorithm form the major part of virtually all kinds of arithmetic circuits. Since arithmetic components often dominate the critical path this makes our approach applicable in all circuits containing arithmetic datapaths.

5. Experiments

For all experiments we used the design environment of Magma Design Automation, Inc.. Our optimizer is implemented as a set of Tcl scripts. For all the other tasks the corresponding Magma functions have been used. Our new design flow consists of the following steps:

1. Module generation
2. Global placement
3. Optimization by operand swaps using the layout data obtained from step 2.
4. Incremental placement with integrated optimization, global and detailed routing

If tight timing constraints were applied the module generation step created nearly balanced Wallace tree-like adders as a starting point. The wire length values used for gain calculation in step 3 have been estimated using the half perimeter bounding box method. The conventional design flow we used for comparison performed the same steps except step 3. For benchmarking we generated circuits for the following expressions and mapped them onto a 0.13µm standard cell library:

1. $y_{[16]} = a \cdot b + c \cdot d + e \cdot f + g \cdot h$
2. $y_{[32]} = a \cdot b + c \cdot d + e \cdot f + g \cdot h$
3. $y_{[16]} = (a+b) \cdot (c+d)+(e+f) \cdot (g+h)$
4. $y_{[32]} = (a+b) \cdot (c+d)+(e+f) \cdot (g+h)$
5. $y_{[32]} = a \cdot b$
6. a 16-bit microprocessor

In our first experiments our goal was to minimize cycle time. The results are shown in Table 1. Columns 2 and 3 contain the improvements in cycle time and area that have been achieved compared to the results of the conventional flow. Both cycle time and area are measured after detailed routing. Hence, they are very accurate and realistic. The area can be different as step 4 of the flow above still performs a lot of optimizations, such as restructuring, buffering and gate sizing.

Circuit	Exp. 1 : min. cycle time		Exp. 2 : min. area
	Cycle time	Area	Area reduction
1	-31%	+2%	-41%
2	-32%	-10%	-37%
3	-2%	-9%	-11%
4	-3%	-15%	-13%
5	-20%	+5%	-18%
6	-15%	+4%	-3

Table 1: Cycle time improvement

The results show improvements in cycle time of up to 32% while the area roughly stays the same. Note that the benchmarks with the largest improvements (1,2) are typical for multiply/accumulate-expressions that occur frequently in signal processing.

In the next experiment we targeted area minimization for a given cycle time. This was done by using the cycle time that could be achieved by the conventional flow as delay target. Now the optimizations in step 4 above had to add less area in order to achieve the performance goal. The area improvements are shown in Column 3.

Finally, we investigated the ability of our algorithm to adapt a circuit to non-uniform input data arrival times. For this purpose we generated a few sets of non-uniform data arrival times for circuit no. 2 randomly. The performance improvements now ranged up to 45% depending on how the data arrival times had been chosen.

6. Conclusion

In this paper we exploit the advantages of an arithmetic bit level representation when optimizing datapaths. This level preserves information about arithmetic symmetries in addition networks that are very expensive to identify using Boolean techniques. Further we presented an algorithm for optimizing circuits containing embedded addition networks in terms of timing. The algorithm considers all addition networks simultaneously to globally optimize the critical path.

The addition networks targeted by the algorithm form the major part of many arithmetic circuits and very often are part of the critical path of large designs. This makes our approach useful in all circuits containing arithmetic parts not only for the multiply accumulate structures on which it performs best.

As the number and type of cells is unchanged by the algorithm it can be applied late in the toolflow. Even on a placed circuit the timing can be optimized without changing the placement. This allows using very detailed timing information to achieve the most accurate results.

In our experiments we obtained performance improvements for complex arithmetic expressions of up to 32% for uniform data arrival times and of up to 45% for non-uniform data arrival times.

References

[1] "Logic Synthesis and Verification", ed. by Hassoun S. and Sasao T., Kluwer Academic Publishers, 2002 Boston/Dordrecht/London

[2] Oklobzija V., Villeger D., Ravi R., "A Method for Speed Optimizing Partial Product Reduction and Generation of Fast Parallel Multipliers using an Algorithmic Approach", IEEE Trans. on Comp. Vol. 5, No. 3, 1996

[3] Stelling P., Martel C., Oklobzija V., Ravi R., „Optimal Circuits for Parallel Multipliers, IEEE Trans. on Comp., Vol. 47, No. 3, 1998

[4] Um J., Kim T., "An Optimal Allocation of Carry-Save-Adders in Arithmetic Circuits", IEEE Trans on Comp., Vol. 50, No. 3, pp. 215-233, 2001

[5] Kutzschebauch T., Stok L., "Congestion Aware Layout Driven Logic Synthesis", Proc. ICCAD-2001, pp. 216-223, 2001

[6] Kutzschebauch T., Stok L, "Regularity Driven Logic Synthesis", Proc. ICCAD-2000, pp. 439-446, 2000

[7] Hartje H., Neumann I., Stoffel D., Kunz W.,Cycle Time Optimization by Timing Driven Placement with Simultaneous Netlist Transformations", Proc. ISCAS-2001, pp. 359-362, 2001

[8] Lou J., Chen W., Pedram M., "Concurrent logic restructuring and placement for timing closure", Proc. ICCAD-1999, pp. 31-35, 1999

[9] Stoffel D., Kunz W., "Equivalence Checking of Arithmetic Circuits on the Arithmetic Bit Level", IEEE Trans. on CAD of Integrated Circuits and Systems, May 2004

[10] Kunz W., Menon P., "Multi-Level Logic Optimization by Implication Analysis", ICCAD-94, pp. 6-13, 1994

[11] Chang C, Marek-Sadowska M., "Perturb and Simplify: Multi-Level Boolean Network Optimizer, ICCAD-94, pp. 2-5, 1995

[12] Sinha S., Brayton R., "Implementation and Use of SPFDs in Optimizing Boolean Networks", ICCAD-98, pp. 103-110, 1998

[13] Chang C., Cheng C., Suaris P., Marek-Sadowska M., "Fast Post-placement Rewiring Using Easily Detectable Functional Symmetries", Proc. DAC-2000, pp. 286-289, 2000

[14] Shin K., Kim T., "An Integrated Approach to Timing-Driven Synthesis and Placement of Arithmetic Cirtcuits", Proc. ASPDAC2004

[15] Neumann I. et al., "Layout Driven Optimization of Datapath Circuits using Arithmetic Reasoning", Tech. Rep. EIS-07-04-01, Univ. of Kaiserslautern, Dept. of Electrical and Computer Engineering, http://www-eda.eit.uni-kl.de, 2004