2009 19th IEEE International Symposium on Computer Arithmetic

Datapath Synthesis for Standard-Cell Design

Reto Zimmermann
DesignWare, Solutions Group
Synopsys Switzerland LLC, 8050 Zurich, Switzerland
reto@synopsys.com

Abstract

Datapath synthesis for standard-cell design goes through extraction of arithmetic operations from RTL code, high-level arithmetic optimizations and netlist generation. Numerous architectures and optimization strategies exist that result in circuit implementations with very different performance characteristics. This work summarizes the circuit architectures and techniques used in a commercial synthesis tool to optimize cell-based datapath netlists for timing, area and power.

1. Introduction

Design Compiler is the Synopsys platform to synthesize RTL code into a standard-cell netlist. Arithmetic operations are first extracted into datapath blocks, undergo high-level arithmetic optimizations and are finally mapped to gates through dedicated datapath generators. This paper highlights the circuit architectures and optimization techniques used to generate optimized netlists for a given design context. Special emphasis is given to the relatively new area of power optimization. The paper does not go into any technical details, but rather gives an overview of what mostly known and published architectures and techniques are successfully applied in our cell-based synthesis flow.

2. Datapath Extraction

Arithmetic operations – such as addition, multiplication, comparison, shift, as well as selection – are first extracted from the RTL code into datapath blocks. Thereby largest possible clusters are formed by including all connected operations into one single datapath block. This allows the subsequent high-level optimizations and number representation selection to be most effective.

The most generic arithmetic operation that is extracted is the sum-of-products (SOP), which also covers simple addition, subtraction, increment, multiplier and even comparison. Therefore the extracted datapath blocks consist of a collection of SOPs, shifters and selectors. An SOP furthermore is composed – just like a simple multiplier – of partial-product generation (PPG), carry-save addition (CSA, Wallace tree) and carry-propagate addition (CPA). All extracted operations can handle inputs and outputs in carry-save number representation, which allows to keep internal results in carry-save and therefore to improve performance by eliminating internal carry propagations [1].

3. Datapath Optimization

After datapath extraction high-level arithmetic optimizations are carried out, such as common subexpression sharing or unsharing, sharing of mutually exclusive operations, comparator sharing, constant folding and other arithmetic simplifications. Sum-of-products are converted to product-of-sums where beneficial ($A*B+A*C \rightarrow A*(B+C)$). Number representations for intermediate results are selected among binary, carry-save and partial-product based on the design constraints and optimization goals. Generally binary representation is used for area optimization and carry-save for delay optimization.

4. Datapath Generation

Flexible context-driven generators are used to implement the gate-level netlist for a datapath under a given design context. This context includes input arrival times, output timing constraints, area and power constraints, operating conditions and cell library characteristics.

4.1. Partial-Product Generation

Partial products have to be generated for multiplication operations. The following generators are included:

1. *Constant multiplication:* **CSD** (canonic signed-digit) encoding of constant input to minimize the number of partial products that need to be summed up.
2. *Binary multiplication:*

1063-6889/09 $25.00 © 2009 IEEE
DOI 10.1109/ARITH.2009.28

a. **Brown** (unsigned), **Baugh-Wooley** (signed): Best architectures for small word lengths (8-16 bits).
b. **Radix-4 Booth**: Good area, best delay for large word lengths.
c. **Radix-8 Booth**: best area for large word lengths, but longer delay.

3. *Special multiplication:*
   a. **Square**: Implemented as *A*A*, bit-level gate simplification and equivalent bit optimization before carry-save addition automatically results in an optimized squarer structure.
   b. **Blend**: Partial-product bits of the blend operation *A*B+~A*C* are generated using multiplexers to reduce the number of product bits.
4. *Carry-Save multiplication:* One input is carry-save, which allows to implement product-of-sums, like *(A+B)*C*, without internal carry propagation. Delay is reduced at the expense of larger area.
   a. **Non-Booth**: Carry-save input is converted to delayed-carry representation (through a row of half-adders), then pairs of mutually exclusive partial-product bits can be added with a simple OR-gate.
   b. **Booth**: Special Booth encoder for carry-save input.

All of the above architectures prove beneficial under certain conditions and constraints. More information and references on the individual architectures are available in [1].

4.2. Carry-Save Addition

All partial-product bits are added up by reducing every column from *N* inputs bits down to 2 output bits (carry-save representation). Column compression is done using compressor cells, such as half-adders (2:2 compressor), full-adders (3:2 compressor) and 4:2 compressors. The **timing-driven adder tree construction** considers input arrival times as well as cell pin-to-pin delays, resulting in delay-optimal adder trees that are better than Wallace trees [2].

4.3. Carry-Propagate Addition

The carry-save output from carry-save addition is converted to binary through a carry-propagate adder. Since carry propagation is a prefix problem, most adder architectures can be categorized as different kinds of *parallel-prefix* adders. These adders share a prefix structure to propagate carries from lower to higher bits. A wide range of prefix structures exists that fulfill different performance requirements. These differ in terms of depth (= circuit speed), size (= circuit area) and maximum fanout (= net loads). The following schemes are used to optimize the adder for a given context:

1. *Parallel-prefix structure:* A prefix graph optimization algorithm is used to generate area-optimized **flexible prefix structures** (unbounded fanout) for given timing constraints [1]. The algorithm is able to generate a serial-prefix structure for very loose constraints, a fast parallel-prefix structure for tight constraints, and a mix of both for constraints in between. It takes bit-level input arrival times and output required times into account and therefore adapts the prefix structure to any arbitrary input and output timing profile, including late arriving single bits or U-shaped input profiles of the final adder in multipliers. This helps reduce circuit area but also delay for the most critical input bits. Bounded fanout prefix structures (like Kogge-Stone) can also be generated, but are not optimized for non-uniform timing profiles.
2. *Sum bit generation:* In the **carry-lookahead scheme** the prefix structure calculates (look-ahead) all the carries, which then compute the sum bits. In the **carry-select scheme** the two possible sum bits for a carry-in of 0 and 1 are calculated in advance and then selected through a series of multiplexers controlled by the carries from all levels of the prefix structure. This latter scheme is used in carry-select and conditional-sum adders.
3. *Prefix signal encoding:* The prefix structure can either compute **generate/propagate** signal pairs using AND-OR gates (optimized into AOI/OAI structures) or **carry-in-0/carry-in-1** signal pairs using multiplexers.
4. *Prefix radix:* The most common parallel-prefix adders use a **radix-2 prefix** structure (i.e., each prefix node processes two input signal pairs to produce one output pair). **Radix-3** and **radix-4 prefix** adders on the other hand process 3 or 4 input pairs in each prefix node. The resulting prefix node logic is more complex but the prefix structure is shallower and thus potentially faster. The high-radix parallel-prefix adders only give better performance if special multilevel AND-OR-gates (like AOAOI) are available in the library.
5. *Special architectures:* **Ling adder** and **spanning-tree adder** architectures are also supported.

All of the above schemes can be combined almost arbitrarily. This allows the implementation of well-known architectures like the Brent-Kung, Ladner-Fisher and Kogge-Stone parallel-prefix adders, carry-select adders, conditional-sum adders, Ling adder, but also many more mixed alternatives.

Most of the described more special adder architectures only give superior performance under rare condi-

tions, if at all. The regular Ladner-Fisher parallel-prefix adder (unbounded fanout, carry-lookahead scheme, generate/propagate signal encoding, radix-2 prefix) with flexible prefix structure gives best and most reliable performance in most cases and therefore is selected most of the time.

4.4. Shifters

Shifters are implemented with multiplexers or AND-OR structures depending on the context. To improve timing the levels inside the shifter can be reordered based on the delay of the shift control inputs. The special case where a shifter actually implements a decoder ($X=1<<A$) is detected and a dedicated decoder circuit is synthesized instead. Shifters can accept binary and carry-save inputs and can therefore reside between two arithmetic operations with no need for a binary conversion (carry propagation).

4.5. Selectors

Selectors are as well implemented with multiplexers or AND-OR structures based on context. They can have arbitrary number of inputs. Selectors also support carry-save inputs, which allows to move them around inside a datapath (e.g., from the binary output to an internal carry-save operand) in order to enable additional sharing.

4.6. Architecture Selection

During synthesis all applicable architectures described in the previous sections are evaluated and costed for timing, area and power. The architecture that fulfills the constraints best for a given cell library is selected. The use of specific architectures can also be manually controlled through dedicated switches.

4.7. Pipelining

Pipelining of datapaths is supported indirectly. All the pipeline registers must be placed at the inputs or outputs of the block in the RTL code. The datapath is then synthesized as a combinational block with a relaxed timing constraint that is proportional to the number of pipeline stages. Finally the pipeline registers are moved to the optimal locations inside the datapath through register retiming.

4.8. Carry-Save Registers

Where a datapath is split by a register (i.e., arithmetic operations on both sides of a register) synthesis can extract the first datapath block with carry-save output and the second with carry-save input, so that the register between them stores an intermediate result in carry-save representation. This optimization, however, currently poses problems in formal verification.

5. Low Power Datapath

Datapaths often consume large amounts of dynamic power due to their large circuit size and high switching activity during operation. Therefore our recent focus was to lower dynamic power dissipation in synthesized datapath circuits.

5.1. Operand Isolation

One of the most effective way to save dynamic power is to turn off entire blocks when not in use. When the output of a block is not read (e.g., deselected by a multiplexer) the inputs into the block are either forced to a constant value or the previous value is kept. Synthesis can automatically apply the former technique and tries to integrate the isolation gates into the datapath logic in order to minimize performance impact.

5.2. Transition-Probability-Based Optimizations

In many applications transition probabilities (switching activities) are not evenly distributed among all inputs. Lower activities on some input operands or some input bits can be exploited to reduce overall activity and power.

5.2.1. Transition-Probability-Based Adder Tree

In applications like signal processing input samples can be correlated and higher bits can have lower transition probabilities. The following techniques from the literature were investigated but found not feasible for cell-based synthesis:

- **Left-to-right multiplier**: Adds the MSB partial products first, which results in lower activity in the entire adder array [3]. However, array multipliers are generally much worse in terms of switching activity and dynamic power compared to tree multipliers because of the much longer signal paths and the ripple structure. Since synthesis usually generates adder trees, synthesized multipliers are already more power efficient than any kind of array multiplier except for very extreme activity distributions.
- **Disable MSB logic**: If upper bits do not change, the upper portion of a multiplier can be shut off [4]. Efficient transistor-level circuits exist for dynamically detecting signal inactivity, but with standard-cells the logic is very expensive. Also, latches would be needed to store previous values, which is problematic in synthesis.

The optimization that was eventually implemented is to balance the adder tree based on input transition probabilities instead of input timing (**transition-probability-driven adder tree construction**). Thereby low activity inputs enter the tree early and high activity inputs late. In the extreme case of monotonically increasing activities from LSB to MSB a structure similar to the left-to-right multiplier is generated. This approach is very flexible and it guarantees an adder tree with lowest possible overall activity for arbitrary input activity profiles.

5.2.2. Transition-Probability-Based Operand Selection

In Booth multipliers the input that is Booth encoded drives extra logic. Power can be saved by Booth-encoding the input with lower average activity. This operand selection is done statically during synthesis. Approaches to dynamically switch operands based on actual activity have also been proposed but are again rejected here due to the too large circuit overhead for activity detection in cell-based design.

5.3. Glitching Reduction

Glitching power, caused by spurious transitions, can be as high as 60% of total dynamic power in datapaths, especially when multipliers are involved. Glitching reduction is therefore considered as the most promising way to reduce dynamic power in datapaths.

5.3.1. Delay Balancing

Glitches are produced when inputs into gates are skewed, i.e., an output transition caused by a transition on an early input is undone later by a transition on a late input. Glitch generation and propagation is largest in non-monotonic gates, like XORs, where each input transition causes an output transition. Preventing input skews through delay balancing can very effectively reduce the amount of produced glitches. It is often sufficient to do it in specific glitch-prone locations only (like Booth encoding/selection), limiting the negative impact of buffer insertion and gate sizing. However, effective delay balancing needs to be done in the back-end (after place-and-route) because only then the final delays are known. It is interesting to note that glitching is usually lower in circuits optimized for speed because skews are smaller when the structure is as parallel as possible and all gates are sized up for optimal speed.

5.3.2. Architecture Selection

Not all architectures are equally prone to glitching. It is observed that in multipliers most glitching originates from partial-product generation, especially in Booth multipliers where signal skews are big because of unbalanced signal paths. A well balanced carry-save adder tree does not produce much additional glitching, but it can propagate incoming glitches all the way through due to the non-monotonic nature of the compressor cells (XOR-based). Thus the adder tree can actually be the largest contributor to glitching power.

Glitching power in non-Booth multipliers can range from 10% to 30%, depending on multiplier size and delay constraints, while for radix-4 Booth multipliers the range is 15% to 60% and for radix-8 Booth multipliers 40% to 70%. Therefore glitching must be taken into account when selecting an optimal architecture for lowest possible power.

5.3.3. Special Cells

Complex special cells, such as **4:2 compressors** and **Booth encoder/selector cells**, can reduce glitching in two ways: a) well balanced path delays can reduce glitch generation and b) long inertial delays can filter out incoming glitches. Synthesis makes use of such cells if available in the library.

A more aggressive way to filter out glitches is to use **pass-gate** or **pass-transistor** compressor cells. Multiple cells in series act like a low-pass filter, but their long delay makes then suitable only for low-frequency applications. However, pass-gate logic is not compatible with current standard-cell methodologies, so this strategy is not considered in synthesis.

5.3.4. NAND-Based Multiplier

Brown and Baugh-Wooley multipliers use $n \times m$ AND-gates to generate the partial-product bits, which are implemented with INVERT-NOR structures for better results ($n+m$ inverters, $n \times m$ NOR-gates). An alternative implementation uses $n \times m$ NAND-gates to generate inverted partial-product bits, which can be added up with a slightly adapted adder tree to generate an inverted carry-save sum. While this architecture does not always give better timing or area, depending on the need for input buffering and output inversion, it often helps reduce glitching because of the absence (of possibly unbalanced) input inverters.

5.4. Shifter Optimization

Dynamic power in shifters is reduced by replacing the multiplexers of a conventional design with demultiplexers, as described in [5]. The resulting structure has lower overall wire load and reduced transition probabilities on the long wires, which both help reduce power dissipation.

5.5. Techniques Incompatible with Cell-Based Design

Other techniques for reducing dynamic power in arithmetic circuits that are found in the literature are

not compatible with cell-based design and synthesis for different reasons. These include:

- *Exchange multiplier inputs* or form *2's complement* of multiplier inputs if this results in lower transition probabilities at the multiplier inputs [6]. The special circuits that are required can be implemented efficiently at the transistor level, but not with standard-cells.
- *Bypass rows/columns* in array multipliers where partial products are 0 [7]. This technique applies to array multipliers and requires latches and multiplexers, which only at the transistor-level can be efficiently integrated into the full-adder circuit.
- *Glitch gating* suppresses the propagation of glitches at certain locations in a combinational block by inserting latches or isolation gates that are driven by a delayed clock [8]. The required latches and delay lines are not compatible with a synthesis flow.
- *Logic styles*, such as pass-transistor logic, can be useful to reduce power because of reduced number of transistors (capacitive load) and through glitch filtering [9], but they are again not feasible in standard-cells.

6. Summary

The extraction of arithmetic operations from RTL into large datapath blocks is key to enabling high-level arithmetic optimizations and to efficient implementation at the circuit level through exploiting redundant number representations and reducing expensive carry propagations to a minimum.

For the synthesis of sum-of-products the different architectures to generate partial products for multipliers all prove to be valuable alternatives to optimize a datapath for a given context. The timing-driven or transition-probability-driven adder tree construction results in optimal carry-save adders for delay, area and power under arbitrary timing and activity input profiles. The flexible parallel-prefix adder with timing-driven prefix structure optimization is the architecture of choice for the implementation of carry-propagate adders and comparators under arbitrary timing constraints.

For the reduction of dynamic power dissipation in datapaths many techniques proposed in the literature are tailored towards full-custom and transistor-level design and therefore are not applicable in cell-based design. On the other hand, the techniques already in place to optimize datapath circuits for area and timing also help improve power because smaller area means less power and the inherent parallelism effectively reduces switching activity and glitching. Only few additional techniques were found that can incrementally reduce power further. Most effective are strategies that help lower glitching power or turn off unused blocks.

7. Outlook

For future work, we see potential for improvements in the following areas:

- Add support for more special arithmetic operations, improve datapath extraction and include more arithmetic optimizations.
- Have more flexible selection of number representations inside datapath blocks, such as partial binary reduction of carry-save results (i.e., propagate carries only for small sections).
- Optimize CSE sharing/unsharing for given timing.
- Investigate more low-power architectures and techniques for cell-based design, also for glitch reduction. Investigate the use of sign-magnitude arithmetic.

8. References

[1] R. Zimmermann and D. Q. Tran, "Optimized Synthesis of Sum-of-Products", *37th Asilomar Conf. on Signals, Systems and Computers,* Nov 9-12, 2003.

[2] V. G. Oklobdzija, D. Villeger, and S. S. Liu, "A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using an Algorithmic Approach", *IEEE Trans. on Computers*, vol. 45, no. 3, March 1996.

[3] Z. Huang and M. D. Ercegovac, "High-Performance Low-Power Left-to-Right Array Multiplier Design", *IEEE Trans. on Computers*, vol. 54, no. 3, March 2005.

[4] K.-H. Chen, Y.-S. Chu, Y.-M. Chen, and J.-I. Guo, "A High-Speed/Low-Power Multiplier Using an Advanced Spurious Power Suppression Technique", *ISCAS 2007: IEEE Intl. Symp. on Circuits and Systems*, May 2007.

[5] H. Zhu, Y. Zhu, C.-K. Cheng, and D. M. Harris, "An Interconnect-Centric Approach to Cyclic Shifter Design Using Fanout Splitting and Cell Order Optimization", *ASP-DAC'07, Asia and South Pacific Design Automation Conference*, Jan 23-26, 2007.

[6] P.-M. Seidel, "Dynamic Operand Modification for Reduced Power Multiplication", *36th Asilomar Conf. on Signals, Systems and Computers*, vol. 1, Nov 2002.

[7] K.-C. Kuo and C.-W. Chou, "Low Power Multiplier with Bypassing and Tree Structure", *IEEE Asia Pacific Conf. on Circuits and Systems, APCCAS 2006*, Dec 2006.

[8] K. Chong, B.-H. Gwee, and J. S. Chang, "A Micropower Low-Voltage Multiplier With Reduced Spurious Switching", *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 2, Feb 2005.

[9] V. G. Moshnyaga and K. Tamaru, "A Comparative Study of Switching Activity Reduction Techniques for Design of Low-Power Multipliers", *IEEE Intl. Symp. on Circuits and Systems, ISCAS '95*, vol. 3, May 1995.