HARVEY MUDD
COLLEGE

Clay-Wolkin Fellowship

Final Report for
Oracle Corporation

Energy-Delay Tradeoffs in CMOS Multipliers

Memo #: CW-10-002

May 16, 2010

Team Members

James Brown
Becky Glick
Matthew Keeter
Ben Keller (Project Leader)
Austin Lee
Andrew Macrae
Daniel Lee (U. Adelaide)
Robert Moric (U. Adelaide)

Advisor

David Money Harris

Liaison

Justin Schauer

Abstract

Oracle Corporation has sponsored a group of students from the Harvey Mudd College Clay-Wolkin Fellowship to develop a methodology by which different digital multiplier designs can be fairly compared. The team developed scripts to generate a variety of different multiplier designs, including array multipliers, Wallace tree multipliers, and 4:2 compressor tree multipliers. The team also evaluated several different tools that could optimally size the gates in each multiplier design: the Stanford Circuit Optimization Tool (SCOT), Large Scale Gate Sizer (LSGS), and Synopsis Design Compiler (DC). The team used each of these tools to generate energy-delay tradeoff curves for different multiplier designs, but was unable to come to very many solid conclusions. In evaluating the optimization tools, the team found significant limitations for SCOT and LSGS, and was unable to obtain sufficiently accurate results using DC. In the end, the DC behavioral synthesis for multiplication had a better energy-delay product than any of the team's designs.

Contents

List of Figures

Chapter 1

Background

1.1 Project Overview

High-speed, low-power digital multipliers are important building blocks both in general computing and in specialized applications such as digital signal processing and cryptography. There are many proposed multiplier designs in the literature, both novel and established [6]. Many of these designs make claims regarding speed and power consumption, but these claims often lack a standardized or fair basis for comparison, making it difficult to compare the energy and delay characteristics of two arbitrary designs in the literature. The goal of this project was to provide such a fair basis for comparison by evaluating many possible designs in a large design space. To perform such fair comparisons, the team employed three different software tools to determine optimal gate sizing under a given energy or delay constraint. Each tool could generate energy-delay (ED) trade-off curves for different multiplier designs. Comparing these curves can provide insight into which designs exhibit favorable energy and delay characteristics.

1.2 Multiplication Overview

Multipliers take in an m-bit multiplicand and a n-bit multiplier and multiply them to produce an $m + n$-bit product. At the heart of many multiplier designs is the generation and summation of partial products to compute the final product. Each of these tasks can be performed many ways, and the final summation is often performed in two stages. The partial product reduction tree performs carry-save additions to sum the partial products

into a pair of outputs. The carry-propagate adder sums the pair to compute the final product. This algorithm is very much like traditional multiplication, and thus multipliers are represented with the partial products in a trapezoidal set of rows so that all the partial products of equal weight form vertical lines. Each partial product can be represented by a set of dots in a dot diagram, with rows that can be generated independently and columns containing equal valued partial products (see Figure 1.1).

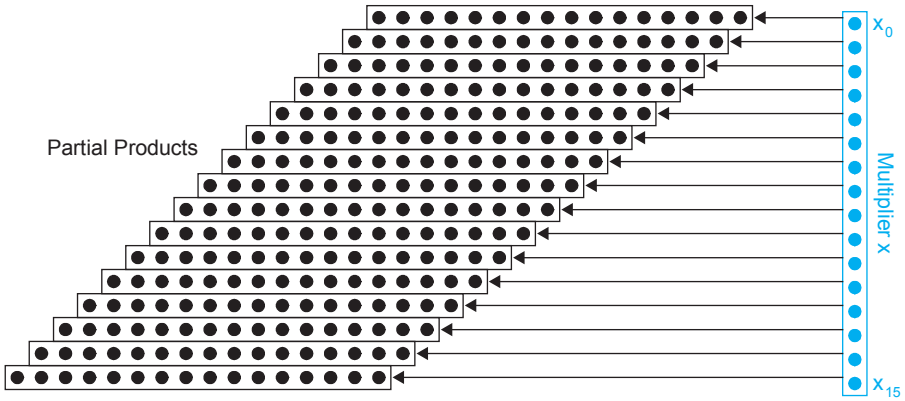


Figure 1.1: A multiplier dot diagram[6]

Multipliers are often discussed and represented with the bitslices (and power and ground lines) running top to bottom, rather than left to right. Fitting the circuit components onto a rectangular block that integrates with standard datapaths requires that the trapezoid be compressed into a square shape.

1.3 Carry Save Addition

Multipliers use full adders to reduce the number of bits to sum in each column. Full adders take in three bits and output their sum as a two bit number. The most significant bit of the sum is called the carry bit; this bit is added to the next column of the multiplier. The least significant bit is called the sum bit; this bit is added within the column from which the bits originated, as it maintains the original weight. Because a full adder takes in three bits, and outputs two bits, it is commonly referred to as a 3:2 carry save adder (CSA). When showing how columns of partial products are added, designers often draw a single column of a dot diagram and then draw CSAs as blocks with three inputs and two outputs. The carry output

is shown to be feeding into the input of a subsequent CSA. Though this is not strictly accurate, each carry that runs into the next column can be considered to be replaced by a carry from the last column. Thus, this is still a decent way to consider the number of bits to sum, and to sketch the placement of the CSAs.

1.4 Design Space

Many sections of a multiplier may be designed independently of each other; that is, the multiplier design space has many different, largely independent dimensions. This implies a very large number of possible design combinations. The team will not be able to test every possible combination of design choices due to the design time involved in generating the HDL code for each multiplier. Therefore, the design space will be explored by branching out from a default design along a number of dimensions. This means that the project won't cover every possible design, but covers much of the sensible design space. The following are brief summaries of each design dimension that the team considered.

1.4.1 Functions

Multipliers can be designed to handle unsigned inputs, signed inputs, or both options in a configurable manner. Early results showed minimal difference between signed and unsigned multipliers, so the team did not pursue this comparison further. All multipliers described below are unsigned.

1.4.2 Booth Encoding

Most high-speed multipliers sum a set of partial products to compute the final product. Booth encoding improves this method further by algebraically manipulating these partial products so that fewer are needed. Radix-4 is a commonly used Booth algorithm that generates only half the number of partial products without introducing significant delay.

1.4.3 Carry Save Adder Style

There are many designs for the cells that perform carry save addition. Depending on the architecture, these designs can improve delay in the sum bit at the expense of the carry bit or change the number of inputs and outputs. Another option is the use of pass transistors to reduce energy, area

and delay. When using SCOT and LSGS, the team was able to perform some comparisons of different types of CSAs; however, the nature of Design Compiler prevented the team from varying this component of the designs in a systematic manner.

1.4.4 Structure

Partial product addition can be improved by arranging the adders in tree structures instead of a linear series because many of the partial products arrive at the same time. This is an area of much research, and there are many choices for how to design a tree. In general, there is a trade-off between shorter wires due to regular layout and lower propagation delay. Therefore, wire capacitance has a major effect on the results since it is the most significant downside to complicated tree structures.

1.4.5 Size

Sizes between 8×8 multipliers and 64×64 multipliers were considered, with different multiplier structures being more or less effective at different sizes.

1.4.6 Final Adder

There are many possible designs for the final carry-propagate adder that sums the two redundant-form outputs of the partial products reduction. The simplest design, a ripple-carry adder, uses little hardware but adds significantly to the delay. Tree-based designs using propagate-generate logic can substantially improve both the delay and the E-D product of multipliers.

1.5 Standard Configuration

Because of the number of dimensions in our design space, it seems overly ambitious to expect to span the space. Instead, the team has selected a base case from which the search can be extended along a range of dimensions to see where particular improvements are attainable. The base case was designed to be simple enough that modifications can be made with relative ease, while realistic enough to serve as a valid basis for comparison. Thus,

the base case is a 16×16 unsigned radix-4 array multiplier that uses uniformly sized XOR3-Majority adders to sum partial products and a Sklansky tree adder to compute the final sum.

Chapter 2

Multiplier Designs

In this chapter, we present the preliminary set of multiplier designs to be examined with the various optimization tools. Each of these designs is created by a Perl generator script. Each script produces a Verilog module that describes the multipliers using cells from a standard library and annotates the Verilog with the positions of the cells.

2.1 Array Multiplier

An array multiplier uses a set of AND gates to compute partial products. Each partial product bit is then fed into a full adder which sums the partial product bit with the sum from the previous adder and a carry from the less significant previous adder. Figure 2.1 shows how the sum and carry propagate through an array multiplier. The structure serves to minimize wire length, produces a regular structure that is easy to lay out, and uses a relatively small area.

In layout out the array multiplier, a rectangular floorplan is desirable. The floorplan is therefore squashed into a fairly regular rectangle. Various extra pieces of hardware and slightly irregular logic make the edges of the rectangle ragged, but the overall shape is highly regular, as seen in Figure 2.2. For more information about the process information relevant to this and future layouts, see Section 4.2.

Figure 2.3 shows that the depth of the CSAs in an array multiplier is N. Array multipliers see limited use at larger sizes due to their lack of tree-based partial products reduction. In this study, they form of base case against which other designs can be compared, and are simple enough to vary other aspects of the design space with the assurance that the overall

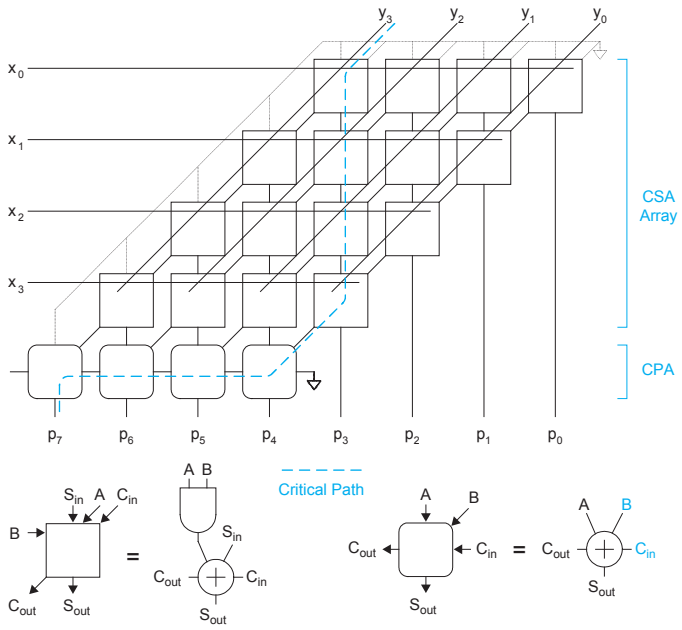


Figure 2.1: Schematic of an array multiplier[6]

design will not be significantly impacted.

2.2 Booth Encoding

Radix-4 Booth encoding increases the complexity of the partial product generation to reduce the number of partial products by a factor of two. Each row of partial products in a booth encoded multiplier represents multiplication by zero through one less than the radix of the multiplier. Thus, a radix-4 multiplier uses a row of partial products to represent the multiplier multiplied by two bits of the multiplicand and represents the multiplier bits scaled by a factor of 0, 1, 2 or 3. Multiplying by 0 or 1 is trivial, and a multiplication by two is an inexpensive shift operation. However, it is difficult to compute the product of a number multiplied by 3 (this is referred to as a "hard multiple"). Booth encoding can be modified to produce negative partial products so that each row is instead multiplied by -2, -1, 0, 1, or 2, and the next row is incremented appropriately to compensate for the negative multiples. This avoids the hard multiple by using the negative output to let two stages represent a 3 as $4 - 1$. With these modifications, radix-4

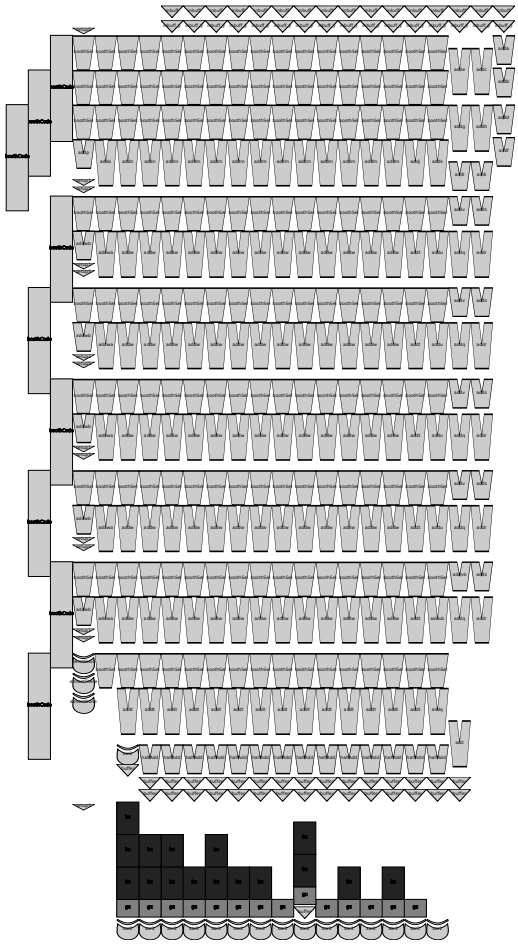


Figure 2.2: Layout of a 16-bit array multiplier

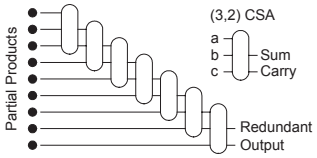


Figure 2.3: Summing of partial products in an array multiplier[6]

modified Booth encoding reduces the number of partial products nearly by half.

2.3 Partial Product Reduction Trees

2.3.1 Wallace Tree

The Wallace tree uses a branching arrangement of CSAs to reduce the set of partial products[22]. This strategy reduces the logical depth, so the critical path grows as $\log_{3/2}(N/2)$, rather than as N as for an array multiplier. The difference in logic depth can be seen by comparing Figures 2.3 and 2.4. Using a Wallace tree to sum partial products should theoretically increase

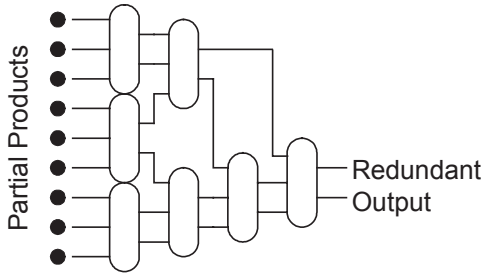


Figure 2.4: Summing of partial products in a Wallace tree[6]

speed compared to an array multiplier, since the critical path of each column has fewer stages. However, the Wallace tree comes with its own set of disadvantages. The tree has an irregular structure, which makes it difficult to lay out efficiently and "squash" into a rectangular floorplan. Furthermore, the tree structure means that results from one block of logic may have to travel a significant distance across the tree to where they are needed, resulting in long wire lengths and performance reduction due to wire capacitance.

As with each design that the team has generated, the HDL for the Wallace tree multipliers are created using Perl scripts. The initial state of the script has a set of partial product rows that need to be summed. In constructing the tree, the script combines sets of three rows with a CSA. The sum and outputs from this CSA are then added into the list of rows to be summed in the next stage of tree reduction. The process repeats until there are only two rows that remain to be summed; these rows are then buffered

and passed into a final adder.

Layout of the Wallace tree is centered on the wordwise regularity: each row is either a partial product generator or a CSA. The tree is organized so that information always flows downwards, and wire length is somewhat optimized. The script places rows in such a way that a row is directly underneath all of its prerequisites. For example, in the first level of tree reduction, each CSA is placed under three rows of partial product generation. This approach to layout keeps the tree structure somewhat regular, as seen in Figure 2.5. In that figure, red indicates level 1 of the Wallace tree, green level 2, blue level 3, and purple level 4.

2.3.2 4:2 Compressor Tree

Taking advantage of a similar branching arrangement to the Wallace tree, the 4:2 tree architecture is constructed from 4:2 compressors. Each compressor receives four inputs and a carry in. It then reduces them to a carry and sum output, as well as a fast carry out. This leads to a total of $\log_2(\frac{N}{2})$ compressors, although each 4:2 compressor has larger delay than a CSA. These compressors can be composed of two 3:2 CSA's, but other designs have been implemented to further optimize the critical path. The carry in and carry out are designed such that the carry out is available after only one XOR delay, while the carry in can be accepted as late as one XOR delay into the two XOR delay computation. Therefore, while 4:2 compressors actually take in five inputs and output three bits, they can be treated as having four inputs and two outputs. Ultimately, the advantage is that 4:2 trees are also much more regular than Wallace trees, reducing wire capacitance and simplifying layout.

Due to time constraints, the 4:2 tree was not completed to the point of a realistic layout.

2.3.3 The Three-Dimensional Method

The three-dimensional method is another partial products reduction technique. Rather than employing a straightforward tree structure, however, it uses a third dimension, signal arrival times, to construct trees that are optimized to minimize delay.

The three-dimensional method takes advantage of the asymmetric gate delay through an XOR-MAJ full adder. The sum bit is calculated by a chain of two XOR2 gates (see Figure 2.6). Note that two inputs must pass through two XOR2 gates to affect the output; for them, the full adder has a delay of

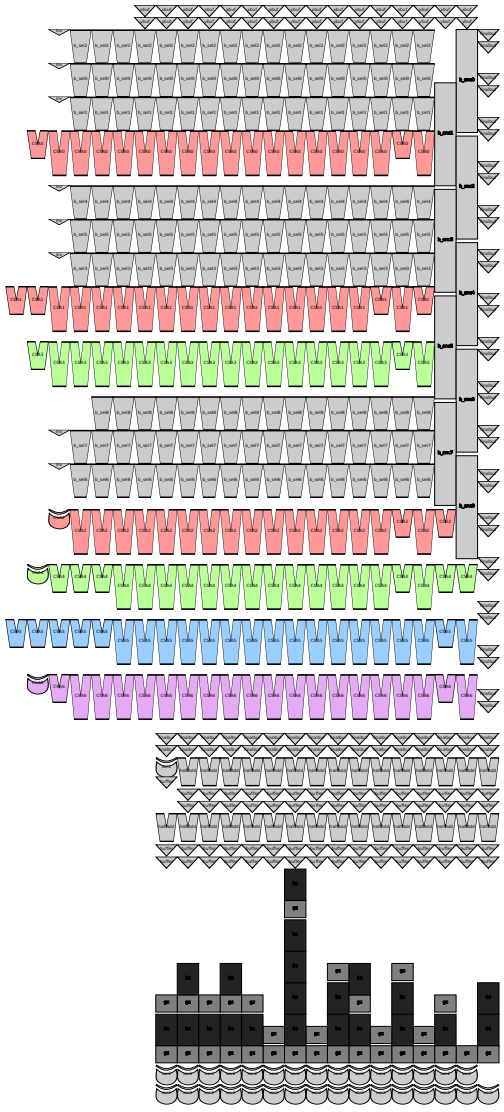


Figure 2.5: 16-bit Wallace tree layout

two XOR delays, the unit in which delay is typically measured for the three-dimensional method. The third input passes through only the final XOR2, and so the sum bit has a delay of only a single XOR gate delay for this input. This means that the third input can arrive later than the first two by up to one XOR delay without affecting the arrival time of the sum bit. The carry bit is symmetric over the inputs; to simplify the three-dimensional analysis, the arrival time of the carry bit is approximated as one XOR delay from the arrival of the last input. It is also necessary to employ half adders for the three-dimensional method. These half adders consist of a single XOR2 gate and an AND gate; therefore, the sum bit has one XOR delay, and the carry bit has approximately half of an XOR delay.

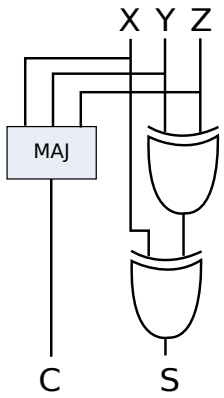


Figure 2.6: An XOR-MAJ CSA [6]

As with other PPRTs, each column of partial products must be reduced to two bits of equal weight, which are then passed to final adder to be summed into the final product. The PPRT that uses three-dimensional method is constructed based on the expected arrival time, in XOR delays, of each intermediate carry signal passed into each column of bits to be summed. As the tree is constructed, the expected arrival times of each carry signal are computed, and passed to the next column.

The algorithm for constructing the reduction tree for each column can be broken into several distinct steps. First, each partial product for the column is accounted for, and assigned an initial delay of zero. Any carry signals from the previous column are also considered; their arrival times are whatever was calculated in the previous column. Note that CSAs compress three input bits into just one output of the same weight, so the total number of equally weighted bits maintains the same parity regardless of

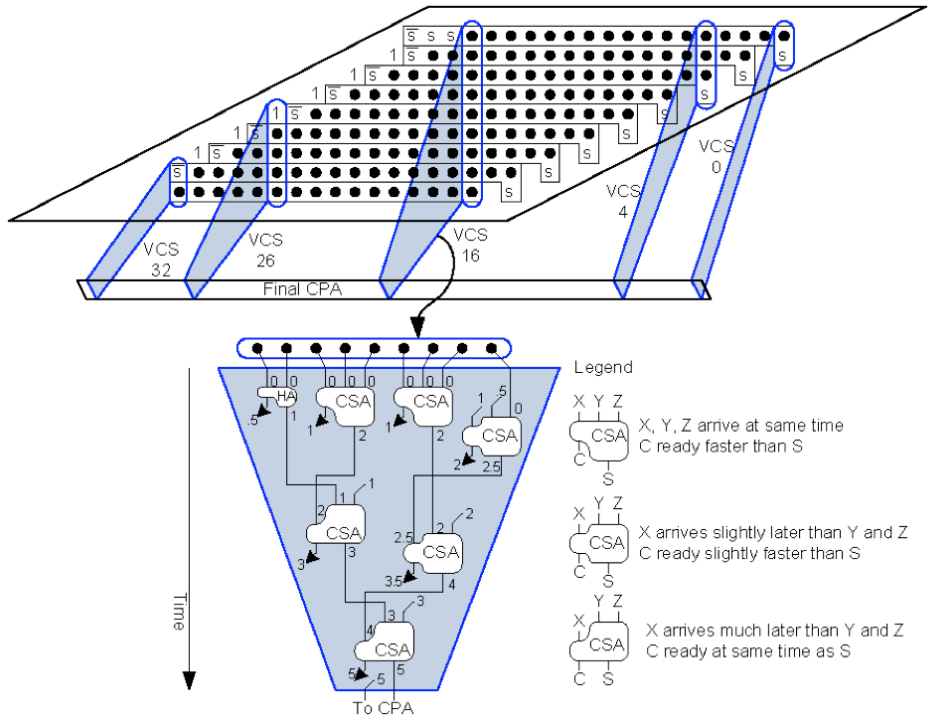


Figure 2.7: The three-dimensional method [6]

the number of CSAs that are employed in the column. Since we wish to reduce the number of bits in the column to two, this cannot be achieved with only final adders if we begin with an odd number of bits. Thus, if the starting number of bits is odd, we begin by passing the two bits with the earliest arrival time to a half adder, which results in an overall even number of bits of the same weight.

The remainder of the tree is constructed using what is referred to as the three-greedy algorithm. This method takes the three partial products or intermediate carry nets with the least delay, and uses them as inputs to the next full adder, arranged so that the signal with the latest arrival time is sent to the fast input of the CSA. The carry output of this full adder is sent to the next column (with the arrival time noted so that the reduction tree of the following column can be constructed), with its delay set to one greater than the maximum of the delays of the input signals. The sum is kept in the current column. Its delay is either one greater than the delay of the fast input, or two greater than the delay of the latest slow input, whichever is

greater. In this way, three bits of equal weight are reduced to one bit in the same column. This process is repeated as necessary until the column has been reduced to two bits of equal weight, which are sent to the final adder. See Figure 2.7 for a graphical representation of the reduction tree for a single column of a 16-bit multiplier.

Although the three-greedy method is not optimal, it is within one XOR delay of optimal for a wide range of multiplier sizes[20]. A TDM multiplier implemented with this the three-greedy method is still faster, in terms of XOR delays, than either 4:2 trees or Wallace trees (see Table 2.1). However, its layout is complicated by its lack of regularity. The added capacitive loading of long wires may significantly increase the overall delay, and the irregular gate structure makes layout difficult.

The layout for the TDM tree was optimized for regularity, at the expense of wire lengths. All of the partial products are generated above, and reduced in a set of adders. The adder rows have been fitted to a rectangular floorplan, so a data line travels downwards and to the right as it passes through the multiplier. The layout can be seen in Figure 2.8.

Number of Partial Products	**Wallace Tree**	**4:2 Tree**	**TDM**
8	8	5	5
9	8	6	6
16	12	8	8
24	14	10	10
32	16	11	11
64	20	15	14

Table 2.1: Comparison of multiplier speeds[6]

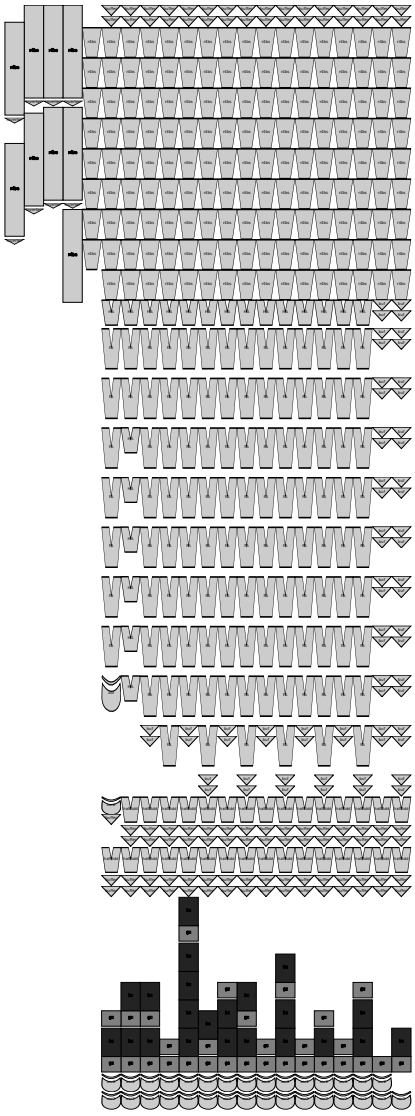


Figure 2.8: Layout of a 16-bit TDM multiplier

Chapter 3

Adder Designs

This chapter discusses final adder architectures and their energy-delay trade-offs. Although a final adder's energy contribution is relatively insignificant compared to the rest of a multiplier, additional delay on the critical path is undesired. Consequently, fast adders can significantly improve multiplier energy-delay products.

We used several fast adders designed by visiting team member Daniel Lee and based off of designs by Sklansky and Zimmerman.

3.1 Sklansky Adder

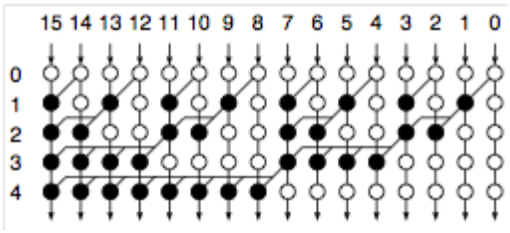


Figure 3.1: PG Diagram of Sklansky Adder[26]

Sklansky adders reduce the number of logic levels in an N-bit adder to $\log_2(N)$ stages by increasing fanout. This is a substantial improvement over a simple ripple-carry adder, the number of logic levels of which increases linearly with the number of intputs. Figure 3.1 is a propagate-

generate diagram of a 16-bit Sklansky adder with dots representing grey and black cells. As a parallel prefix adder, Sklansky uses black cells for group generate and propagate logic and grey cells at the end of each column for generate logic. Assuming uniform arrival times from the multiplier partial product reduction tree, a Sklansky adder is optimized for delay.

A Sklansky adder's sparse structure allow for a vertical fold to compact layout into a smaller rectangle. Figure 3.4 is a visualization of a 32 bit folded Sklansky adder. The light grey colored hardware on the top and bottom of the layout are convert the signals into and out of propagate-generate form.

3.2 Zimmerman Adder

A Zimmerman adder combines the benefits of a ripple carry adder's limited hardware with a Sklansky adder's low delay[26]. Zimmerman adders optimize for non-uniform arrival times with the strategy of making the critical path fast and by then reducing hardware wherever possible.

Figure 3.2 illustrates a Zimmerman adder composed of a ripple carry adder for the first bits which arrive early, and then transitions into a Sklansky adder for the late arriving higher bits.

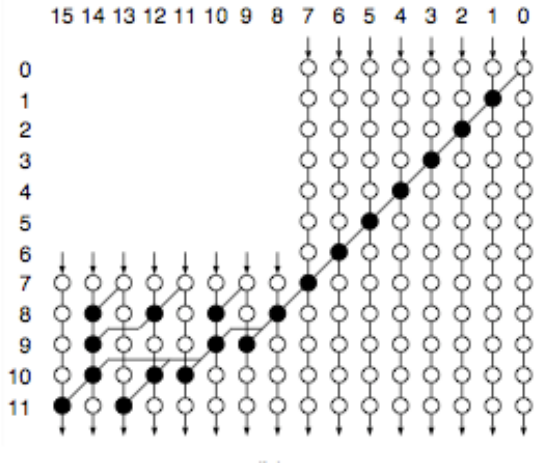


Figure 3.2: PG diagram of a simple Zimmerman adder[26]

More accurately representing a multiplier's final adder, Figure 3.3, optimizes for "trapezoidal" arrival times. A simple ripple carry adds the early

arriving most and least significant bits and a Sklansky passes bits along the critical path.

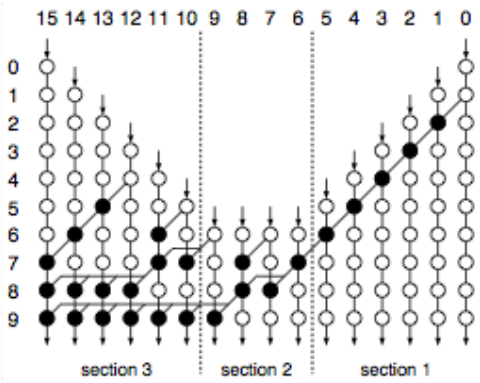


Figure 3.3: PG diagram of a three section Zimmerman adder[26]

A Zimmerman adder layout reflects the architecture's hardware reducing benefits, as seen in Figure 3.5. The team generated adders with this algorithm by estimating the arrival times of each bit to the final adder, an estimate arrived at simply by counting the number of stages in the partial product reduction.

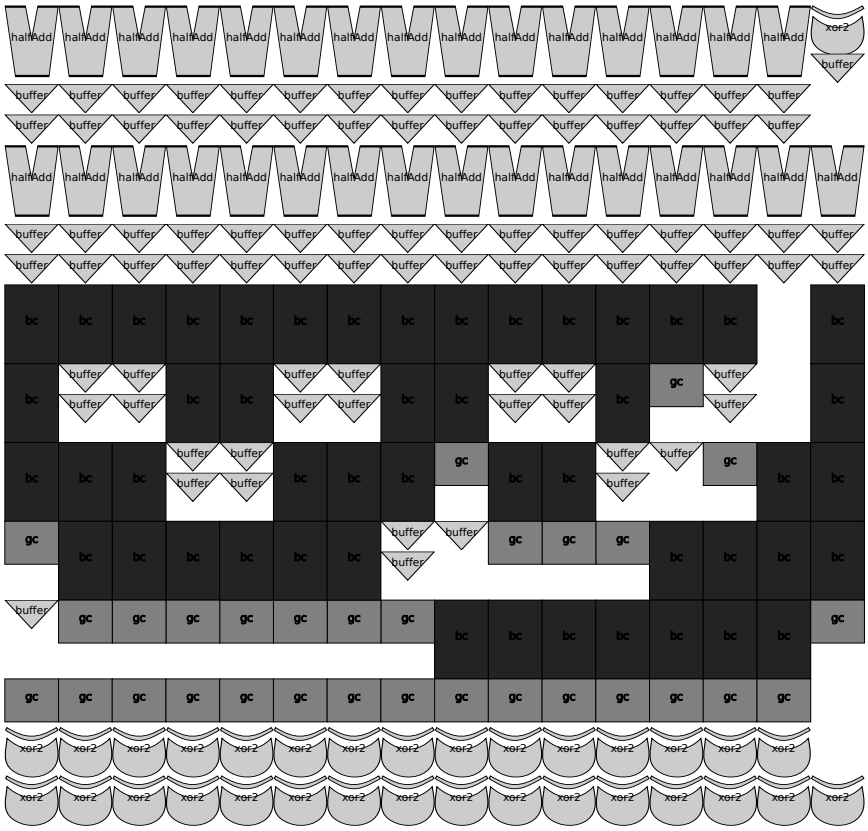


Figure 3.4: Layout of 32-bit Sklansky Adder

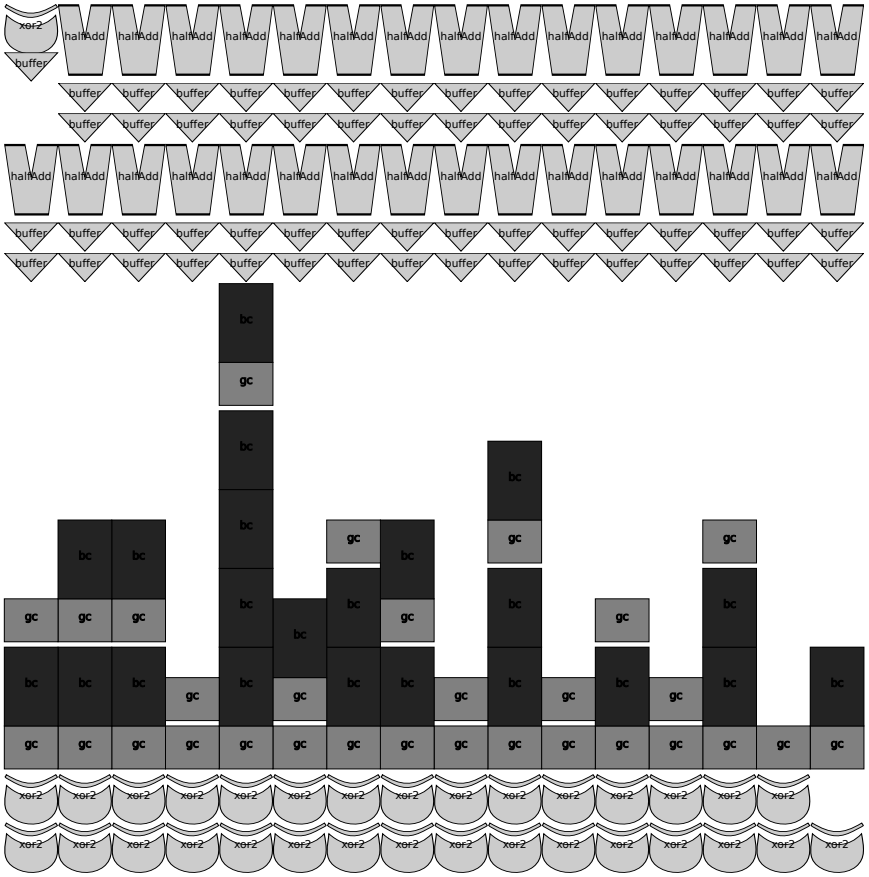


Figure 3.5: Layout of 32-bit Zimmerman Adder for 16-bit TDM

Chapter 4

Optimization

This chapter describes the tools and techniques that are being used to automatically optimize the multipliers. The team employed three different tools to evaluate multiplier performance via sizing optimization: the Stanford Circuit Optimization Tool (SCOT), Large-Scale Gate Sizer (LSGS), and Synopsis Design Compiler (DC).

4.1 Motivation

One of the focuses of this project is to ensure that the results are believable and applicable to industry. The use of optimization tools helps to ensure this goal in two ways. First, it ensures that human design is only needed at the architectural level, while the gate-level or transistor-level sizing is automated. This means that the results are less dependent on the team's skill in designing circuits, and that all of the designs are treated fairly by the same algorithm. Secondly, these tools simulate the circuit using real process models, ensuring that our results are applicable to real chips using recent technologies.

4.2 Process Information

SCOT uses a 90nm model from an ST Microelectronics process with a V_{DD} of 1.1 V and an FO4 delay of 27 ps. SCOT designs assumed a column pitch of 80λ. All cells used in SCOT designs were hand-designed as part of a custom standard library.

LSGS and DC use a 130nm IBM process and an ARM standard cell library, with a V_{DD} of 1.2 V and an FO4 delay of 55 ps. These designs assume a column pitch of 60λ.

Wire capacitance was estimated as $0.2\,^{\mathrm{fF}}/_{\mu\,\mathrm{m}}$ for both processes.

4.3 SCOT

SCOT is a tool for optimizing transistor sizing to minimize either the energy or the delay of a circuit. It works by generating a generalized problem from a circuit that is a purely mathematical description of the optimization problem. This problem is then solved using the MOSEK optimization software in terms of transistor sizing as well as any other desired variables, such as V_{DD}. SCOT can be used to generate E-D trade-offs by applying a delay constraint and then minimizing energy to obtain a single E-D point. This process is then iterated until an entire curve is defined from absolute minimum delay to absolute minimum energy.

4.3.1 SCOT Flow

A Perl script called `auto_iter` was created that takes in a Verilog file and automatically generates an ED trade-off curve. It also has options to enable activity factors, wire capacitance, and uniform sizing. Activity factor calculation is achieved using a ModelSim simulation dump and then back-annotated into the SPICE deck along with the duty cycle. Once it is in the SPICE deck, SCOT automatically uses the data in its calculations. Wire capacitance is calculated using module positions described in the Verilog file. Wire length is estimated for each net using a modified Prim algorithm for rectilinear Steiner trees. These capacitances are then inserted into the SPICE deck for each net. A schematic of the overall project flow is shown in Figure 4.1.

4.3.2 Challenges

Unfortunately, the team found that SCOT could not scale up to multipliers larger than 32 bits. Initially, it couldn't even handle 32-bit multipliers. To attempt to remedy this, options were added to `auto_iter` in an effort to decrease the number of free variables in the optimization problem.

The first method of speeding up SCOT was to implement uniform gate sizing. This decreases the computation time involved in the optimization

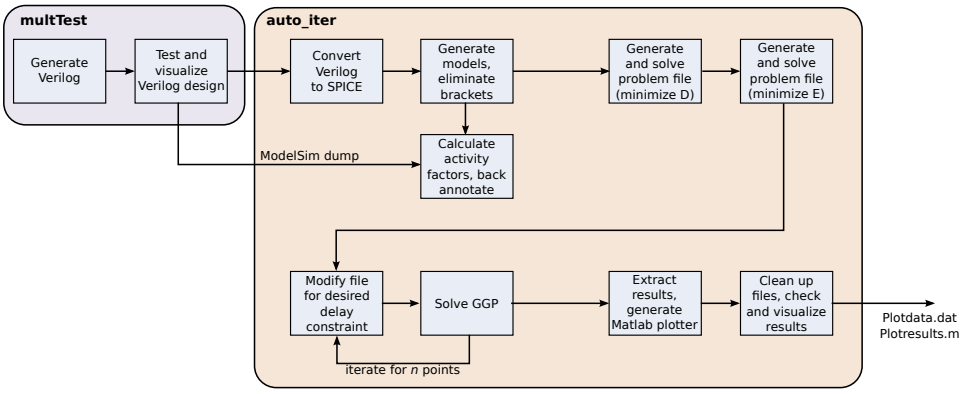


Figure 4.1: Project flow with SCOT

problem by setting modules of the same hierarchy to have shared transistor size variables. For example, if uniform sizing is enabled, SCOT would force all NANDs in all CSAs to be sized similarly, although NANDs outside of CSAs could be sized independently from those in the CSAs. This not only improves the execution speed, but also realistically simulates how an actual multiplier would be laid out with regularity.

Uniform sizing was able to decrease the problem size from thousands of free variables down to tens of variables, but it wasn't enough to get the larger multipliers working. To further speed up SCOT, all transistors in a cell were set as fixed ratios of a single variable. This required more design to be done because these ratios had to be pre-set, but it made 32 bit multipliers possible in SCOT. With uniform sizing and fixed transistor ratios together, only 6 variables need to be solved in any size multiplier.

Although we expected huge speed increases from these changes, we only ended up getting moderate gains of doubling the speed at best. This is likely due to the fact that SCOT still needs to generate a large model for the entire multiplier and isn't able to simplify the problem even when a single variable is used in many equations. In addition, the effectiveness of the optimizer is reduced by these changes, making the results more dubious and farther from optimal. For these reasons, the team decided to switch to LSGS in order to handle larger multipliers.

4.4 LSGS

The Large-Scale Gate Sizer (also known as "LSGS" or simply "Gatesizer") is a tool designed specifically for the task at hand — optimizing medium-to-large circuits. It is being developed at Sun Labs by Jo Ebergen. It works by transforming the gate sizing problem into a convex approximation, which can be solved using standard methods[7] very efficiently.

As opposed to SCOT's exponential performance, LSGS is linear in the number of gates in the circuit (and with a rather small constant). However, after some time working with LSGS, we determined that it would not be feasible for this project due to some modeling issues, which are discussed in Section 4.4.2.

4.4.1 LSGS Flow

LSGS is divided into a number of components, some of which are shown in Figure 4.2.

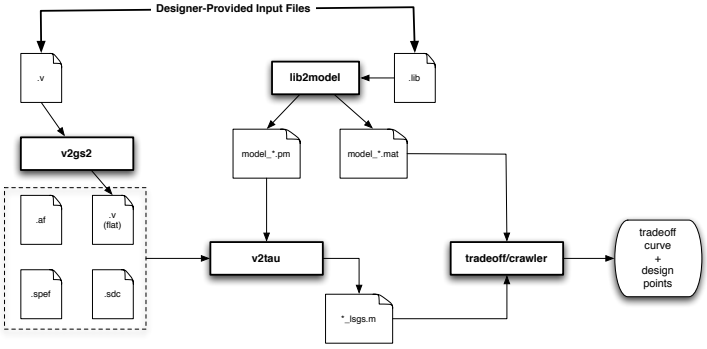


Figure 4.2: Project flow with LSGS

We interact with LSGS through two in-house scripts, `gs_gen` and `gs_run`. `gs_run` only exists to parse user commands and call `gs_gen`. `gs_gen` is responsible for taking the files provided (typically, Verilog for a multiplier) and preparing them for LSGS.

The first stage is to convert the Verilog into a form that LSGS can understand. This requires flattening the input down to a single level and converting our gates into those present in the ARM library, which is done by `v2gs2`.

From this flattened file, activity factors and wire capacitances are generated as described in Section 4.3.1. An SDC file is also generated which

provides the constraints for LSGS. In particular, it specifies the input and output loads on the multiplier. The team is currently using x4 inverters for both driving cells and output loads.

Ben Keller and Austin Lee later investigated using automatic routing tools to extract more precise wire capacitance values for LSGS[9], and found that the difference between estimated and extracted capacitance values did not have a significant impact on delay.

Characterizing the library is a completely separate process, as indicated in the diagram. It is handled by a script named `lib2model`, which parses the Liberty file (particularly its rise/fall response curves) to generate the simplified delay and energy information used by LSGS. It writes this out both as Perl and Matlab files, since LSGS contains components written in both languages.

Once all of the intermediate files have been generated, they are passed to `v2tau` to create a Matlab data file for LSGS's use. This is then passed to either `tradeoff` or `crawler`, Matlab scripts. `tradeoff` generates an energy-delay curve by taking designer-provided maximum- and minimum-delay points, then sampling points in between and optimizing to meet the relevant constraints. `crawler` takes a rough estimate minimum-energy point and then hill-climbs to the minimum-delay point. In our project, we primarily used `crawler`.

Once these MATLAB scripts are run, `gs_run` passes their output to another script which assembles output graphs as shown in Figure 4.3.

4.4.2 LSGS Issues

LSGS, as a work-in-progress research tool, has a number of limitations which eventually led the team to abandon it for this project. The most severe of these is a flaw in its gate models. In this standard library, many cells are implemented as a multi-stage design. It makes the choice to decrease drive delay in larger-sized cells by only increasing the output stage. This causes upsized cells to have substantial intrinsic delay increases. Unfortunately, LSGS does not model this size-dependent intrinsic delay at all, which leads to nontrivial modeling problems. This is illustrated in Figure 4.3, which displays the results of an LSGS run on an 8-bit ripple-carry adder.

In this graph, the blue curve shows the output of LSGS's internal (unsnapped) modeling, and the black diamonds are the results of Primetime simulations at particular snapped instantiations. The slope of the Primetime curve is actually the opposite of what LSGS expects it to be, due to

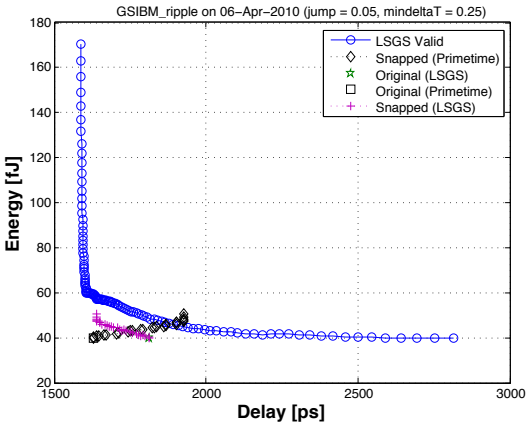


Figure 4.3: LSGS results for ripple-carry adder

.

this unmodeled intrinsic delay increase. This problem is most evident with chained compound gates (like full adders), and thus makes using LSGS with multipliers challenging.

We also encountered a number of other bugs in LSGS. For example, it does not characterize gates with very asymmetric rise/fall delays very well, it can't properly simulate gates which have multiple inputs being driven by the same signal, and it has problems determining starting conditions.

4.5 DC

After the difficulties with LSGS, the team chose to investigate using Design Compiler (DC) to process the designs. Design Compiler (DC) is an industry standard tool by Synopsys. It performs synthesis of Verilog files using a provided library of cells—in this case, the ARM library that we used for LSGS tests. DC produces energy and delay information using Primetime for timing information and its own internal model for energy calculations. Though DC preserves logical function, it modifies the code in various ways to optimize the performance of the circuit. For example, it is much more adept at taking advantage of the asymmetric timing properties of the standard cells to optimize delay along a path.

The flow through DC is shown in Figure 4.4. The Verilog and constraints file are generated as before. Then, the Verilog is optimized with

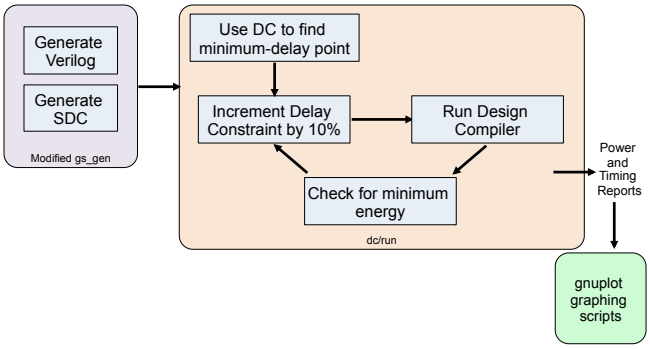


Figure 4.4: Project flow with Design Compiler

DC for a range of delay constraints. The resulting power and delay are visualized by a set of automatic scripts that parse, filter, and plot the data. The graphs can be seen in Chapter 5 of the report. It should be noted that this flow does not take into account our earlier layouts and does not currently include any wire capacitance estimation.

The team also attempted to combine DC and LSGS; that is, use LSGS to by DC. Again, the graphs can been seen in Chapter 5 of this paper.

4.6 Visualization Tool

The team is using manually-laid-out Verilog to more easily account for wire capacitance and other geometrically-dependent effects. The team has developed a visualization tool named `cwviz` to verify layouts and visually compare different multiplier designs. This tool is designed to render Verilog with custom positioning tags to a SVG (Scalable Vector Graphics) file and, when necessary, automatically convert to other formats. It is implemented in Ruby and, wherever possible, is configurable without any coding knowledge. A sample visualized cell is displayed in Figure 4.5. The visualizer currently has configuration files for displaying cell-type colorization, no colorization, and images.

In the second semester, `cwviz` was extended to also visualize SCOT output. It can both display absolute per-cell timing (which is good for finding the slowest or fastest cells in the circuit), or "relative" timing, which makes it easy to see the critical path. The tool also has support for doing some preliminary layout checking for a circuit, including detecting cell overlap. An 8×8 irregular TDM multiplier with relative timing is shown in Figure 4.6.

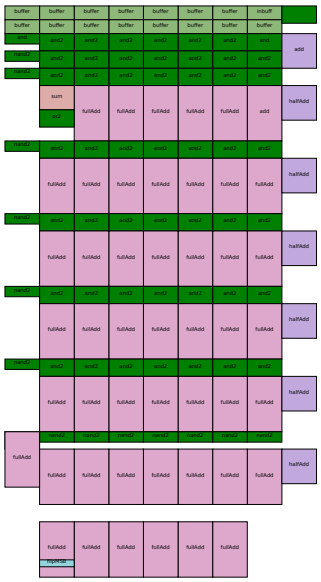


Figure 4.5: 8-bit array multiplier visualized with `cwviz`

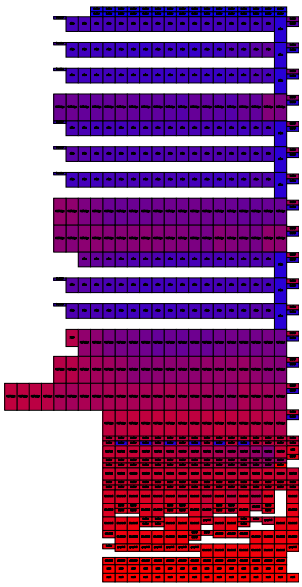


Figure 4.6: 16-bit multiplier visualized with delays

Chapter 5

Results

In this chapter, we present a set of results in the form of energy-delay tradeoff curves. Using automated optimization tools (as described in Chapter 4), we are able to build multipliers with various energy and delay characteristics and create energy-delay tradeoff curves, which we show and explain in this section.

5.1 Validity

For each optimization strategy, an effort was made to ensure that optimizations were producing useful data. Delay estimation was performed using logical effort and counting XOR delays; power estimation was performed by estimating total capacitance and average activity factors. At all stages, we compared between different optimization methods and tools as sanity-checking.

5.1.1 Validity of SCOT Results

Our validation for SCOT involved testing the critical path to check that the delays met our estimations. Analysis of FO4 inverter chains matched previous characterizations. Multipliers demonstrated delays similar to those predicted by counting XOR delays. As we sought to speed up the runs, we further constrained the problem, despite penalties to the validity of the results. For more information, see Section 4.3.2.

5.1.2 Validity of LSGS Results

Because LSGS relies on gate level simulation, the optimization problem is more heavily constrained than the one solved by SCOT. When comparing LSGS results to those from SCOT, it is important to note that LSGS simulations are performed a different process that is roughly half as fast as the process used with SCOT. Verification of LSGS results involved testing inverter chains against predictions made with logical effort, as well as a number of simple cells (including NAND trees and a ripple carry adder). With the exception of the ripple carry adder, LSGS modeled systems as expected. Due to tool issues explained in Section 4.4.2, results from circuits including chains of compound gates (such as a ripple-carry adder) are invalid; we are thus not using LSGS results except in very well-controlled circumstances.

5.1.3 Validity of DC Results

Design compiler is a commercially available tool. We were not able to run optimizations with wire capacitance, but the other tools showed that the effects of wire capacitance were not significant enough to prevent tree structure adders from outperforming array multipliers for large sizes. However, the lack of wire capacitance estimation means that comparisons between the tree-based multipliers are questionable.

5.2 SCOT findings

The transistor-level optimization strategy employed by SCOT helped us form some interesting conclusions. Transistor-level simulation let us explore how limiting the degrees of freedom in an optimization problem penalized the performance of the multiplier, as well as how wire capacitance affected different TDM layouts.

5.2.1 Gate-Sizing Strategies

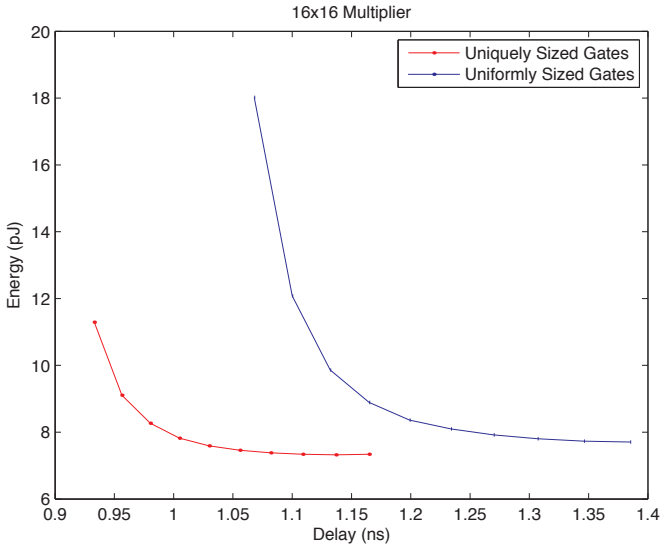
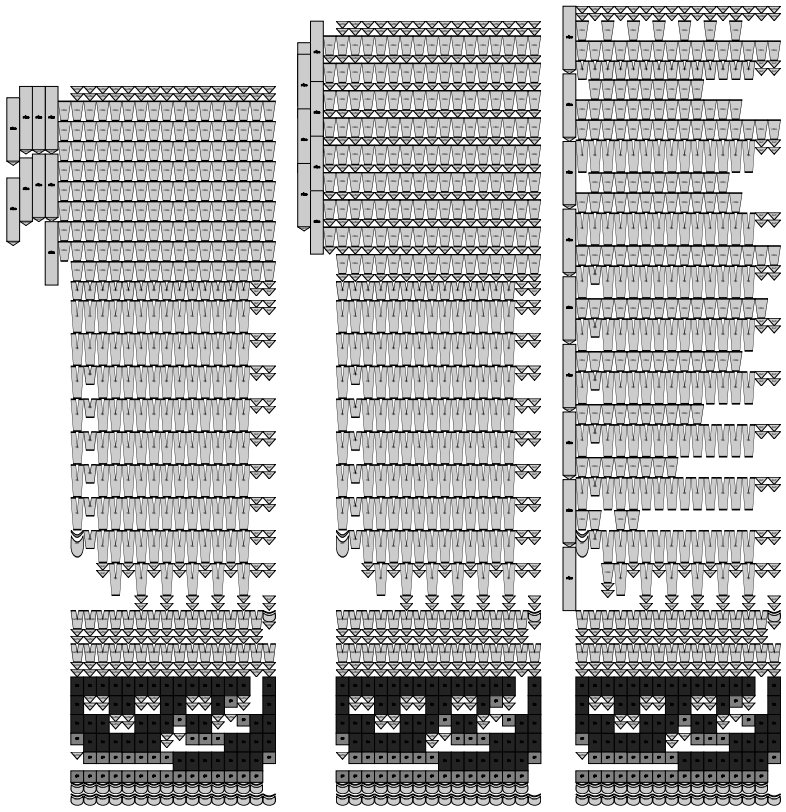


Figure 5.1: Performance penalty for uniform sizing in a 16-bit TDM

While trying to reduce the time SCOT needed to optimize a multiplier, we experimented with a number of strategies to simplify the optimization problem posed to SCOT. One method we examined was to size similar gates uniformly. Essentially, this would mean that every instance of a given gate in the circuit (for example, every inverter) would be sized identically. Figure 5.1 shows a test-case for a TDM multiplier. Note that this constraint increased the delay by about 20%. This demonstrates that to produce valid results with SCOT, we would need to use transistor-level sizing, which we found to be impractical at sizes above 16 bits.

5.2.2 TDM Layout Design



(a) regular (b) regular (buffered) (c) smooshed

Figure 5.2: Three different 16-bit TDM multiplier layouts

Because SCOT was aware of how layout affected multiplier speed, several layout strategies for a TDM multiplier were tested in order to find a layout which could best deliver the performance gains promised by the TDM architecture. Figure 5.2(a) shows a TDM multiplier that achieves a regular structure by separating the partial product generators from the tree adders on the left side. Partial products and adders are generated on a diagonal, but the diagonal has to jog left in order to reach the tree adder. Figure 5.2(b) shows a multiplier in which buffers have been added before these long wires in an attempt to speed the transmission of partial products. This should be useful in large multipliers and on long wires that may have to travel a reasonably large vertical distance in addition to the hori-

zontal distance. Figure 5.2(c) shows a different strategy in which the partial products are generated within the tree adder where they will first be used. This design sacrifices some regularity and lengthens some wires in order to reduce the total number of long wires.

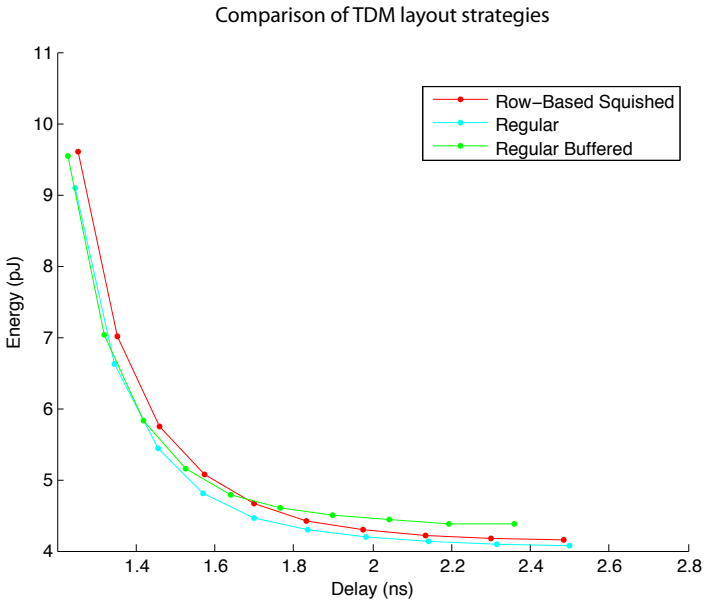


Figure 5.3: ED tradeoff curves for three TDM multiplier layouts

SCOT results for each of these layouts are compared in Figure 5.3. Note that for a 16-bit multiplier, the buffering only consumes extra power and does not affect delay significantly. Also note that the lengthening of a few long wires for the "smooshed" design (Figure 5.2(c)) dominates the delay and causes the smooshed TDM to perform more slowly and to require more energy than the regular structure. The design shown in Figure 5.2(a) had the best energy-delay tradeoff curve, and thus was used for all further experiments.

5.3 DC Findings

We found that Design Compiler was well suited to our optimization toolchains. Because it used Primetime for its model of delay, it was able to perform optimizations that performed especially well in Primetime simulations. Design compiler did not respect our net-lists, and often took advantage of several oddities in our standard cell libraries that we still do not fully understand, but by doing so it's pretty clear it came closer to any optimum than LSGS.

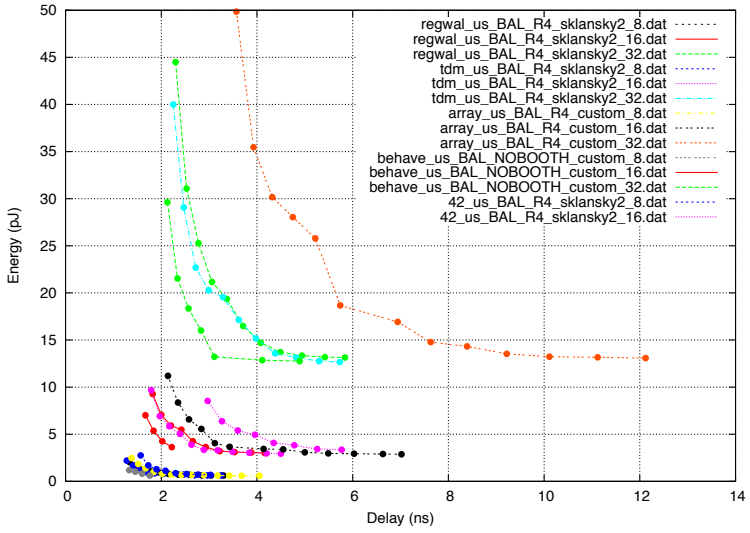


Figure 5.4: ED tradeoff curves for various architectures at various sizes

To date, the team has tested 8-bit, 16-bit, 32-bit and 64-bit multipliers. Figure 5.4 shows how size causes energy to increase as the square of the size, while delay's increase is dependent on the architecture. We limited our investigation to multipliers with only numbers of bits that were an even power of two.

5.3.1 Architectures

Multipliers were built and compared with five different architectures and four sizes.

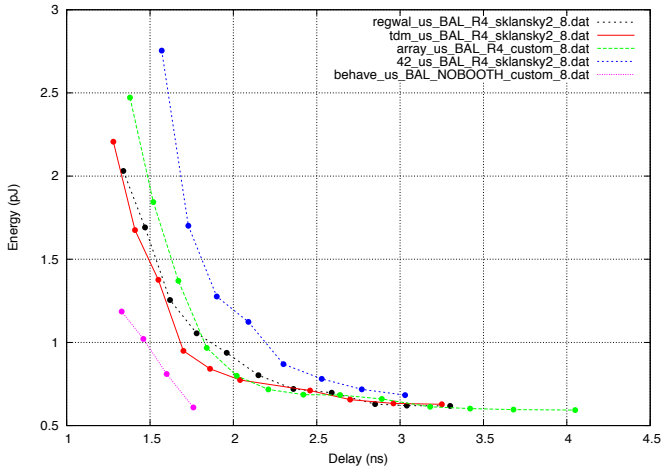
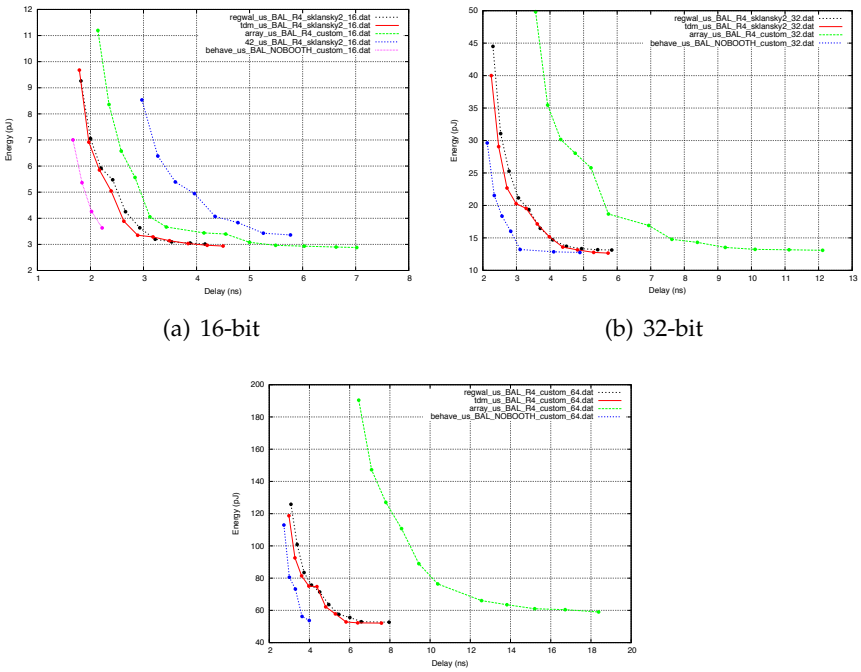


Figure 5.5: ED tradeoff for 8-bit multipliers of various architectures

The team has compared various architectures in order to see the real effect of their differences on the design tradeoffs. The Wallace, TDM, and 4:2 multipliers are implemented with Sklansky final adders while the array multiplier is implemented with Zimmerman adders.

Figure 5.5 shows how our implementation of various designs compare in energy usage and delay at 8-bits. These simulations are all performed without examining effects of capacitance. The 4:2 tree seems to have a mislaid critical path which prevents the realization of the performance gains we would expect from a tree structure. At small sizes, the array performs comparably to the tree multipliers. The curve with the lowest E-D product represents the output from Design Compiler when generating a module from pure synthesis (the tool was programmed simply to synthesize `assign y=a*b`). Design Compiler appears to use a 4:2 tree without Booth encoding for low power operation, and a Wallace tree with Booth encoding for high speed operation.

Figure 5.6(a) shows how the implementation of various designs compare in energy usage and delay for 16-bit multipliers. Note that as the size of the multipliers increases, the array multiplier begins to slow down; this is expected due to its asymptotically-linear performance. The behaviorally



(a) 16-bit

(b) 32-bit

(c) 64-bit

Figure 5.6: ED tradeoff curves at 16, 32, and 64 bits

synthesized curve still outperforms the team's designs, though it did not demonstrate lower energy operation.

Figures 5.6(b) and 5.6(c) show how our implementation of various designs compare in energy usage and delay for 32- and 64-bit multipliers. Note that the 4:2 compressor-tree design is not included for any size above 16 bits.

5.3.2 Booth Encoding

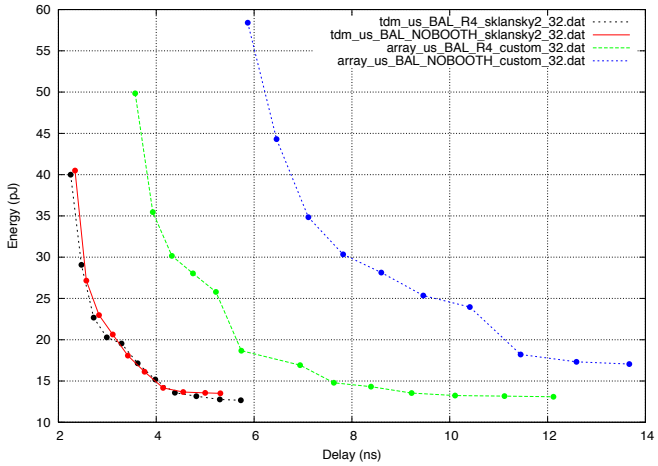
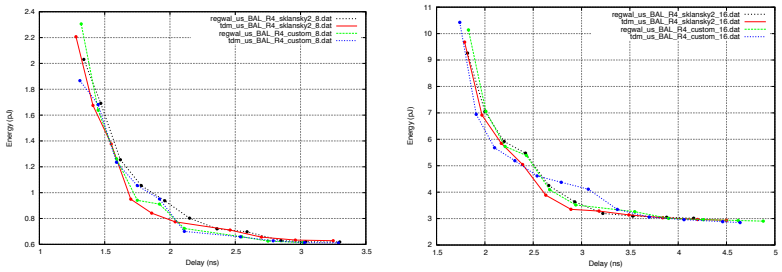


Figure 5.7: ED tradeoff curves for radix-4 Booth-encoded and non-Booth-encoded multipliers

Booth encoding reduces the number of partial products. Figure 5.7 shows its benefits for the 32-bit array multiplier. It's worth noting that in all the other structures, Booth encoding improves the area required for the multiplier, but does not significantly change the energy or delay because the time required to perform the encoding and compute the partial products is roughly equal to the time saved by removing the stages in the partial product reduction tree. The nature of these tree-based structures is such that a large reduction in the number of partial products results in only a small reduction in the number of logic levels in the tree.

5.3.3 Final Adders



(a) 8-bit (b) 16-bit

Figure 5.8: ED tradeoff curves using Sklansky and Zimmerman final adders

Using parallel prefix adders improves the runtime of the multipliers significantly. Figure 5.8 compares Sklansky adders to Zimmerman adders and shows that the improvements offered by optimizing the final adders are insignificant. Figure 5.8(b) also shows the same comparison at 16 bits and shows that the Zimmerman adders do not significantly outperform Sklansky adders.

5.3.4 Asymptotic Performance

The assumption that minimum delay is linear for array multipliers and logarithmic for tree multipliers was tested and is supported by our findings. Figure 5.9 shows the trend for minimum delay with increasing multiplier size. The fits shown are linear and logarithmic. Equation 5.1 describes the minimum delay of a TDM multiplier in terms of FO4 delays. Equation 5.2 describes the minimum delay of an array multiplier in terms of FO4 delays.

$$\text{delay}(N) = 10\text{FO4} \times \log_2(N) - 7.6\text{FO4} \tag{5.1}$$

$$\text{delay}(N) = 1.6\text{FO4} \times N + 12.3\text{FO4} \tag{5.2}$$

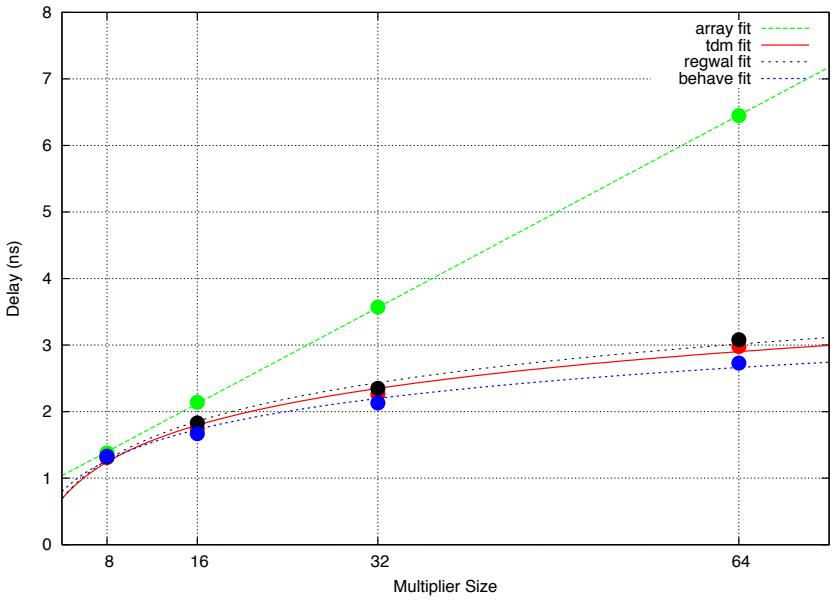


Figure 5.9: Asymptotic behavior of minimum delay for tree and array multipliers

Chapter 6

Conclusions

The initial goal of this project was to use SCOT to compare the energy-delay properties of different multiplier designs. The team did not complete this objective, but was instead able to evaluate several different optimization tools while still drawing some useful conclusions about multiplier design.

The team was able to evaluate the practicality of several different optimization tools for optimizing energy or delay of relatively large circuits. The team attempted to use SCOT, LSGS, and Synopsis Design Compiler to complete this objective.

The team believed that the energy-delay optimizations from SCOT were accurate, but SCOT was limited by run-time considerations. The team took several steps to limit the number of variables that SCOT was attempting to optimize, including restricting all gates of the same type to the same size. These changes did result in speed improvements, but they were not substantial enough to make SCOT feasible as a general optimization tool for large circuits. Specifically, 16-bit multipliers were generally feasible to run in less than 24 hours on the research server, but larger circuits took weeks to complete or exceeded the available memory on the server.

Large Scale Gate Sizer never suffered from runtime issues, but failed to produce credible results on any circuits composed of more than a few gates. In short, LSGS oversimplified the optimization problem, failing to account for the effects of multiple-output gates, imbalanced rise-fall times, and various other minor bugs. As a tool still in development, LSGS may still have potential for use in applications with large circuits, but its current form does not produce reliable results for multiplier circuits.

Synopsis Design Compiler also does not face runtime issues, and as an industry standard, its results can be trusted. Unfortunately, DC is some-

thing of a black box, and does not respect the team's cell positions, gate definitions, or even netlists. The lack of physical placement information in the team's DC flow meant that the DC results could not credibly address the layout differences that are key to distinguishing different tree-based multipliers. Additionally, Design Compiler was able to synthesize a behaviorally-defined multiplier that outperformed the team's designs, suggesting that the team's choice of gates was not taking advantage of the diverse array of cells in the ARM standard library and that the team's designs therefore were not credible.

Each of the three optimization tools has applications in circuit design, but their use in optimizing multipliers was not suitable for the goals of the multipliers project.

Appendix A

Deliverables

All of our code is stored in an SVN repository on `chips`, the VLSI lab's Linux server. This repository is accessible from anywhere on the Claremont Colleges via the URI `svn+ssh://chips.eng.hmc.edu/srv/svn/mult`. Important directories are listed below:

dc/ Scripts for interacting with Synopsys Design Compiler.

final/ This report

gatesizer/ Scripts for interacting with LSGS

midyear/ The mid-year report

notes/ Miscellaneous documentation

perl/ Multiplier generation scripts

poster/ The project's poster

presentations/ Presentations about the project

scot/ Scripts for interacting with SCOT

tools/ Miscellaneous scripts (e.g., test runners)

tv/ Test vectors

verilog/ Generated verilog

Where appropriate, `readme.txt` files have been added to explain the contents of individual directories.

Bibliography

[1] G. W. Bewick. *Fast Multiplication: Algorithms and Implementations*. PhD thesis, Stanford University, 1994. Number CSL-TR-94-617.

[2] D. Crookes and M. Jiang. Using signed digit arithmetic for low-power multiplication. *Electronics Letters*, 43(11):613–614, May 2007.

[3] H. Dhanesha, K. Falakshahi, and M. Horowitz. Array-of-arrays architecture for parallel floating point multiplication. In *Advanced Research in VLSI, 1995. Proceedings., Sixteenth Conference on*, pages 150–157, March 1995.

[4] H. El-Razouk and Z. Abid. Area and power efficient array and tree multipliers. In *Electrical and Computer Engineering, 2006. CCECE '06. Canadian Conference on*, pages 713–716, May 2006.

[5] H. Eriksson, P. Larsson-Edefors, M. Sheeran, M. Sjalander, D. Johansson, and M. Scholin. Multiplier reduction tree with logarithmic logic depth and regular connectivity. In *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, pages 4–8, 2006.

[6] D.M. Harris and N. Weste. *CMOS VLSI Design*, chapter 10. Pearson Education, Inc., 2010.

[7] S. Joshi and S. Boyd. An efficient method for large-scale gate sizing. *IEEE Transactions on Circuits and Systems I*, 55(9):2760–2773, October 2008.

[8] J.-Y. Kang and J.-L. Gaudiot. A fast and well-structured multiplier. In *Digital System Design, 2004. DSD 2004. Euromicro Symposium on*, pages 508–515, August 2004.

[9] B. Keller. Synthesized routing and parameter extraction of multipliers. Technical Report CW-10-003, Harvey Mudd College, 2010.

[10] S.-R. Kuang, J.-P. Wang, and C.-Y. Guo. Modified Booth multipliers with a regular partial product array. *Trans. Cir. Sys.*, 56(5):404–408, May 2009.

[11] D. Kudithipudi, P. Nair, and E. John. On estimation and optimization of leakage power in CMOS multipliers. In *Circuits and Systems, 2007. MWSCAS 2007. 50th Midwest Symposium on*, pages 859–862, August 2007.

[12] N. Mehmood, M. Hansson, and A. Alvandpour. An energy-efficient 32-bit multiplier architecture in 90-nm CMOS. In *Norchip Conference, 2006. 24th*, pages 35–38, November 2006.

[13] Z.-J. Mou and F. Jutand. A class of close-to-optimum adder trees allowing regular and compact layout. In *Computer Design: VLSI in Computers and Processors, 1990. ICCD '90. Proceedings., 1990 IEEE International Conference on*, pages 251–254, September 1990.

[14] 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 Transactions on Computers*, 45(3):294–306, March 1996.

[15] D. Patil, O. Azizi, M. Horowitz, R. Ho, and R. Ananthraman. Robust energy-efficient adder topologies. In *Computer Arithmetic, 2007. ARITH '07. 18th IEEE Symposium on*, pages 16–28, June 2009.

[16] B.C. Paul, S. Fujita, and M. Okajima. A ROM based low-power multiplier. In *Solid-State Circuits Conference, 2008. A-SSCC '08. IEEE Asian*, pages 69–72, November 2008.

[17] M.R. Santoro and M.A. Horowitz. SPIM: a pipelined 64×64-bit iterative multiplier. *Solid-State Circuits, IEEE Journal of*, 24(2):487–493, April 1989.

[18] P.-M. Seidel, L.D. McFearin, and D.W. Matula. Secondary radix recodings for higher radix multipliers. *Computers, IEEE Transactions on*, 54 (2):111–123, February 2005.

[19] M. Sjalander and P. Larsson-Edefors. High-speed and low-power multipliers using the Baugh-Wooley algorithm and HPM reduction tree. In *Electronics, Circuits and Systems, 2008. ICECS 2008. 15th IEEE International Conference on*, pages 33–36, September 2008.

[20] P.F. Stelling, C.U. Martel, V.G. Oklobdzija, and R. Ravi. Optimal circuits for parallel multipliers. *IEEE Transactions on Computers*, 47(3): 273–285, March 1998.

[21] J.-H. Tu and L.-D. Van. Power-efficient pipelined reconfigurable fixed-width Baugh-Wooley multipliers. *IEEE Transactions on Computers*, 99 (1):1346–1355, 2009.

[22] C.S. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, EC-13(1):14–17, February 1964.

[23] M.-C. Wen, S.-J. Wang, and Lin Y.-N. Low power parallel multiplier with column bypassing. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, volume 2, pages 1638–1641, May 2005.

[24] J.-H. Yen, L.-R. Dung, and C.-Y. Shen. Design of power-aware multiplier with graceful quality-power trade-offs. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, volume 2, pages 1642–1645, May 2005.

[25] S. Zhou, B. Yao, J.-H. Liu, and C.-K. Cheng. Integrated algorithmic logical and physical design of integer multiplier. In *ASP-DAC '05: Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 1014–1017, New York, NY, USA, 2005. ACM.

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

[27] D. Zuras and W.H. McAllister. Balanced delay trees and combinatorial division in VLSI. *Solid-State Circuits, IEEE Journal of*, 21(5):814–819, October 1986.