Design Methodology of Regular Logic Bricks for Robust Integrated Circuits

Kim Yaw Tong, Vyacheslav Rovner, Lawrence T. Pileggi, and Veerbhan Kheterpal[†]
Carnegie Mellon University, and Fabbrix Inc.[†]
{ktong, vrovner, pileggi}@ece.cmu.edu, and veerbhan@fabbrix.com

Abstract*—Regularity in IC design has been recognized as an effective means to combat variability in nanoscale technologies. One way to enforce design regularity is to implement ICs using a small library of** regular logic bricks. **In this paper we propose a methodology for the design and synthesis of such logic bricks. Since logic bricks are comprised of a limited set of logic primitives for manufacturability reasons, we propose a primitive-based direct mapping approach for generating optimized bricks that, in contrast to classical synthesis approaches, can provide direct control of implementation structures at abstract functional level based on the detection of natural decompositions that exist in the function. We demonstrate considerable improvement in the performance of logic bricks that are generated by the proposed method as compared with those produced by a commercial synthesis tool.

***Index Terms*—circuit design, manufacturability, regularity, synthesis**

I. Introduction

WITH scaling of CMOS technologies to 65nm and below, IC designs are suffering from decreasing yield, increasing variations, and escalating design and manufacturing costs. The root cause of these problems stems from the fact that the wavelength of light used in lithography steps has not been keeping up with technology scaling. Process engineers have been introducing Resolution Enhancement Techniques (RETs) to mitigate the discrepancy between the scaling rate of design features and the wavelength of light used to print them. Nevertheless, faithful reproduction of patterns over the range of defocus and exposure latitudes in photo-lithography has become highly dependent on layout neighborhood. This has led to hotspots (e.g. poly shorts), line-end shortening, and poor Across Chip Line-width Variation (ACLV) which result in systematic yield losses [1].

It is widely recognized that physical regularity leads to better control of variability, as exemplified by the use of specialized sub-DRC rules in the design of SRAM bit-cells [2]. However, enforcing shape-level regularity in ordinary logic has been challenging since generic logic tends to be irregular, and the added constraints to produce physical regularity can result in substantial area-delay penalty. Regularity in IC design has received extensive attention in recent years as a means to better control variation during manufacturing. Various forms of regular designs have been proposed, including restrictive design rules [3], PLA-based fabrics [4], homogeneous gate-array-like fabrics [5], and heterogeneous regular fabrics [6].

Fundamentally, logic elements in standard cells libraries are too fine-grained to be efficient building blocks of regular ICs, while homogeneous logic elements that are configurable to perform various logic functions tend to be underutilized [7]. Consequently, regular ICs built from application/domain-specific coarse-grained regular logic blocks could be the most efficient. In this paper, we investigate a method to generate coarse-grained logic building blocks that produce efficient implementations while facilitating various manufacturability benefits afforded by regular ICs. In particular, we present a primitive-based brick generator that produces logic bricks compatible with those reported in [6] and [1].

We begin the remainder of this paper with a summary on the benefits of regular logic bricks in Section II. In Section III, we provide an overview of a proposed design flow for regular ICs constructed with bricks. Following that, we focus on the primitive-based brick generator in Section IV. Finally, we present experimental results in Section V, and our conclusions in Section VI.

II. Benefits of Regular Logic Bricks

A regular logic brick is a small logic function implemented physically by mapping onto a micro-regular layout fabric. The boundaries of logic bricks contain similar logic primitives to provide a common geometrical interface between any two bricks. In terms of logic granularity, bricks are more coarse-grained than simple standard cells as the logic function performed by each brick is more complex than a standard cell. Using logic bricks, robust IC designs that are based on a small number of RET-friendly regular geometry patterns can be constructed.

This work was funded in part by an IBM PhD Fellowship, an Intel Fellowship, and the SRC under contract number 2005-HJ-1325.

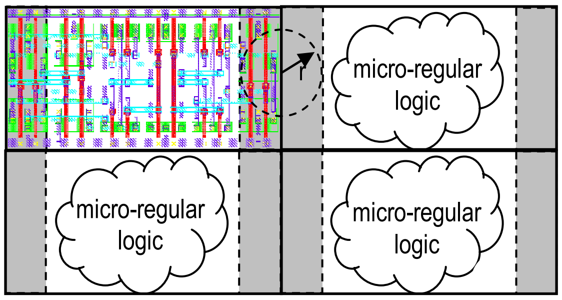

Figure 1. Logic bricks mapped onto regular fabric.

A. *Regularity by Construction*

By defining a highly regular layout fabric on which logic bricks are implemented, optical proximity correction (OPC) is simplified and manufacturability is improved by the reduced number of geometry patterns and lithography interactions that must be analyzed. The exact layout fabric is technology specific, though key features include unidirectional fixed width lines placed at a constant pitch or a multiple thereof. For example, the fabric in Figure 1 utilizes polysilicon and metal1 in vertical orientation and metal2 in horizontal orientation. In addition, the fabric explicitly prohibits certain patterns that are known to have a poor process window [1]. Logic bricks mapped onto a regular fabric exhibit regularity by construction. Layouts adhering to the fabric guarantee that the implemented design will have a sufficient process window.

B. *Efficient Circuit Level Implementation*

Another benefit that coarse-grained logic bricks have over standard cells is that very efficient circuits can be implemented within the logic bricks. For example, it is well known that pass-transistor logic (PTL), when interleaved with static CMOS, is highly efficient in implementing many functions [8]. However, such efficiency might not be reaped in a standard cells approach because the multiplexers (Muxes) in standard cells are buffered at inputs and output due to uncertain loads in standard cells environment. On the other hand, since the exact electrical environment is known within a logic brick, unnecessary buffers or inverters can be removed. This certainty of a well-characterized environment is in fact a crucial criterion to enable the efficient use of PT/static logic within bricks. It is worth noting that the ability to remove unnecessary inverters not only reduces design area, but also has significant impact on leakage reduction as inverters are the most problematic structures in terms of leakage.

C. *Efficient Layout*

While the benefits of regular layout fabrics could potentially be applied to a standard cells library, the area penalty associated with implementing designs using a regular-fabric-compliant standard cells library would be too great due to the fine-grained nature of standard cells. Similar to circuit level efficiency, the extra flexibility afforded within coarse-grained logic bricks enables more efficient layouts. Since the sizes of regular logic bricks are on the order of several standard cells, we can gain substantial area reduction by using diffusion sharing between common nodes.

As a simple example, consider the implementation of function Z = ab + c’ using a pair of Nand2 gates. A standard cells layout of such a structure is shown in Figure 2 (a). Classic standard cells placement tools would leave space between adjacent cells as they have no notion of sharing common nets across standard cell boundaries. In contrast, a brick layout takes advantage of that information to obtain a more compact layout. Specifically, we have used the fact that diffusions of transistors sharing a common net can be abutted to reduce the implementation area (by about 20% in this example) as shown on Figure 2 (b).

III. OVERALL DESIGN FLOW

In order to effectively utilize the efficient circuit level implementation and the layout compactness of logic bricks, we propose a design flow which directly maps a RTL netlist into bricks. A typical design would require a small library of less than 20 logic bricks. The contents of the brick library are derived such that it enables an efficient implementation of the target design. Figure 3 shows the synthesis and optimization flow of the derivation of a brick library for a given RTL netlist.

Large Boolean nodes in the RTL are factored and decomposed into smaller nodes, each of which has a support size in the range of 5 to 10 inputs. Each of these small Boolean nodes can be directly implemented as a logic brick. This is in contrast to conventional two-step procedures where a technology-independent synthesis is followed by a technology binding step [9]. In such methods, the technology binding step is constrained by the netlist structure produced during the technology-independent step. Further, due to a large technology library, the technology binding step is complex and is difficult to model during technology independent optimization.

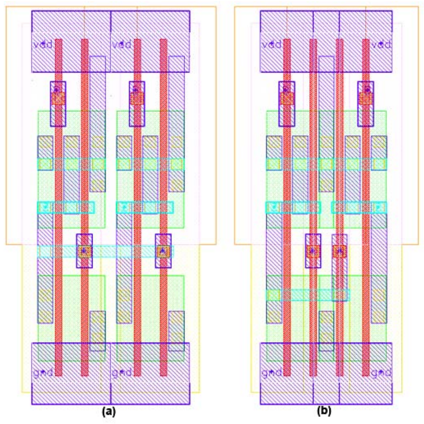

Figure 2. (a) Standard cells, and (b) brick layouts.

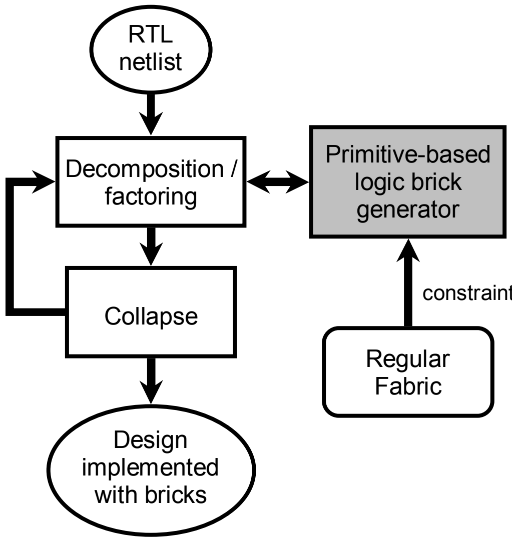
RTL netlist
Decomposition / factoring
Primitive-based logic brick generator
Collapse
constraints
Regular Fabric
Design implemented with bricks

Figure 3. RTL to logic brick synthesis.

In the proposed flow, the efficiency of each brick implementation drives the factoring and decomposition decisions. Consequently, the brick generator provides an accurate estimate of area-delay for the factors. Further, the netlist is collapsed and re-decomposed iteratively in order to minimize the number of distinct logic bricks produced, thereby minimizing the number of unique geometry patterns in the IC implementation.

IV. PRIMITIVE-BASED BRICK GENERATION

The aforementioned material in Sections II and III provides some background for the Regular Logic Brick methodology. The remainder of this paper focuses on our Primitive-Based Brick Generator (PBBG). PBBG leverages the tractable and well-characterized environment within logic bricks to perform efficient logic and circuit level optimizations. Unique features of PBBG include: 1) it utilizes structures of BDDs to guide functional decompositions, and 2) there exists a one-to-one correspondence between decompositions and the logic primitives used to implement bricks.

BDD structures are used to identify the most natural and efficient decompositions in the logic functions, while the correspondence between decompositions and logic primitives enables abstract functional decompositions to directly control physical implementation and allows electrical issues to be taken into account during decompositions.

Figure 4 shows a brief overview of PBBG. The procedure begins with an abstract Boolean function and iteratively decomposes the function from output toward inputs. During every iteration, an abstract function node is selected, and a collection of possible decompositions are evaluated. The decomposition that produces the lowest cost estimate is selected, and the corresponding primitive is used to decompose the function. The factors from this decomposition are inserted into the abstract function queue if they are not implemented yet. The iteration stops when there is no more abstract function in the queue.

In [6], Kheterpal et al. have chosen a logic primitive set composed of Nand2, 2:1-Mux, and inverters. In this paper, we expanded the primitive set to include Nand3, Nor2, Aoi21, Oai21, Aoi22, and Oai22. The purpose of this expansion is to provide a primitive set that contains the most frequently used logic elements which appear in logic bricks to enable the most efficient implementations.

A. Logic Decomposition and Electrical Characterization

We employed a decomposition scheme that uses structures of BDDs to guide the decompositions. The basic theory of BDD-based logic synthesis was presented by Yang and Ciesielski in [10]. By analyzing the structures of BDDs, decompositions that occur naturally in the functions can be found. One attractive feature of this scheme is that it can effectively identify natural Mux-based decompositions in addition to And/Or-based decompositions. Nevertheless, efficient use of Muxes as discussed in Section II.B requires electrical analysis to justify the use of unbuffered Muxes. Figure 5 shows SPICE simulation results verifying that all our logic primitives can safely drive an unbuffered Mux of fanout-of-one in designs where slew constraint is set to be 60ps.

Building upon the theory presented in [10], we have developed a set of functional decompositions that directly correspond to the logic primitives we use to build logic bricks. In addition, we have developed a novel procedure to identify common sub-factor of two functions by analyzing their BDDs. As shown in Figure 6, two functions, G and H, that share a common sub-factor, K, will also share a sub-BDD. By identifying such a sub-BDD, we can decompose G and H in terms of K. Namely, $G = T_G(X_G, K)$ where X_G is a subset of

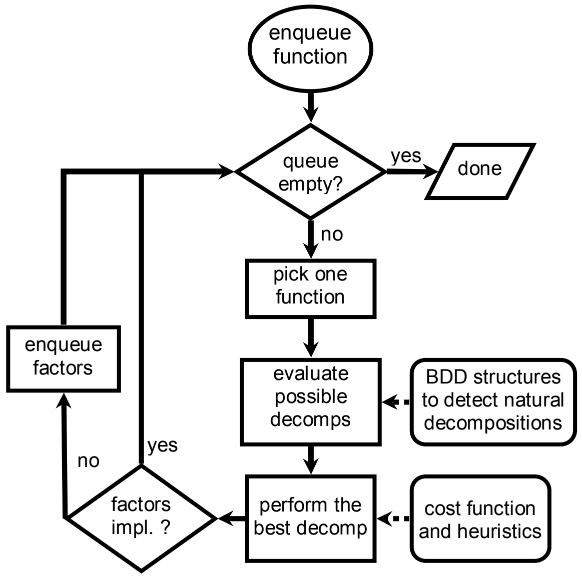

Figure 4. Overview of PBBG.

the original support of G, and T_G is the decomposition function given by $T_G = k'G|_{K=0} + kG|_{K=1}$ for a new intermediate variable k = K. H is also decomposed in a similar manner so that the sub-factor K is shared by both G and H. Notice that such a sub-factor detection method is only feasible in PBBG by virtue of the small size of the logic functions.

We have also implemented the *interval_cofactor* algorithm presented by Stanion in [11] rather than the *restrict* operator used by Yang for don't care minimization. Our experiments showed that Stanion's algorithm produced logic bricks of better quality than those produced using the *restrict* operator.

More importantly, Yang's work focused on large Boolean networks, and used BDD decompositions as a technology-independent logic optimization technique. It has been reported that some optimality is lost in the subsequent technology binding step. In contrast, due to the one-to-one correspondence between logic primitives and decompositions in PBBG, there is not a separate technology binding step in our flow. Consequently, abstract logic level decompositions have direct impact on the actual physical implementation of the logic bricks. Again, this is only feasible for a small set of logic primitives, as imposed in the Regular Logic Brick methodology for manufacturability reasons.

B. *Cost Function and Heuristics*

Given that Boolean functions are directly mapped onto logic primitives during functional decomposition, it is essential for the brick generator to have an accurate estimate of the final implementation cost throughout the decomposition procedure. At the beginning of the process we only have an abstract Boolean function. In the middle of the process, we will have a partially implemented Boolean network with some of the nodes being actual logic primitives, while other nodes remain as abstract functions. Therefore, we need a good cost function to accurately correlate abstract Boolean functions with their eventual implementations.

We have formulated the following cost function, and have experimentally verified that it does give a very good estimate of the efficiency of the final implementation.

$$\text{cost} = \alpha * \text{area\_cost} + (1\text{-}\alpha) * \left(\text{sup\_cost} * \sum \text{fact\_cost}\right)$$

The various components of the cost function are as follows: sup_cost is a measure of the amount of support reduction achieved in the decomposition. It is defined as

$$\text{sup\_cost} = \frac{\sum(\text{factors support size})}{(\text{original function support size})}$$

For a bi-decomposition (i.e. one that produces exactly two factors), sup_cost is a number between 1 and 2, where 1 indicates a disjoint decomposition (i.e. the supports of the factors do not overlap at all), while 2 indicates a complete overlap of the supports. Intuitively, a disjoint decomposition is preferred because at least one of the factors is guaranteed to have its support size be reduced to half of the original function's support size.

fact_cost is zero if the factor has already been implemented, but if the factor has not already been implemented, then fact_cost is computed as follows:

$$\text{fact\_cost} = \sum \text{nodes}(\text{factor}) + \sum \text{work}(\text{factor}) + \text{diff}(\text{factors})$$

where nodes(factor) returns the number of nodes in the BDD representing the abstract factor and work(factor) returns the computational work [12] of the factor.

$$\text{work}(\text{factor}) = 2^n * \text{entropy}(\text{factor})$$

where n is the support size of the factor and entropy(factor) is the informational entropy measure of the factor.

Finally, the term diff(factors) captures the absolute difference in the estimated costs of the factors. The purpose of this term is to avoid decompositions that produce highly imbalance factors. In the case of bi-decomposition, a decomposition with low diff(factor) and low sup_cost will produce factors that *both* have support sizes halved relative to the original (i.e. the decomposition is balanced and disjoint).

In addition to the cost function, we also employed a set of heuristics based on good circuit design practices. These heuristics include using good drivers (e.g. Inverter, Nand2) at the output of a brick, and avoid having cascaded Muxes that require buffering between series drain/source of transistors.

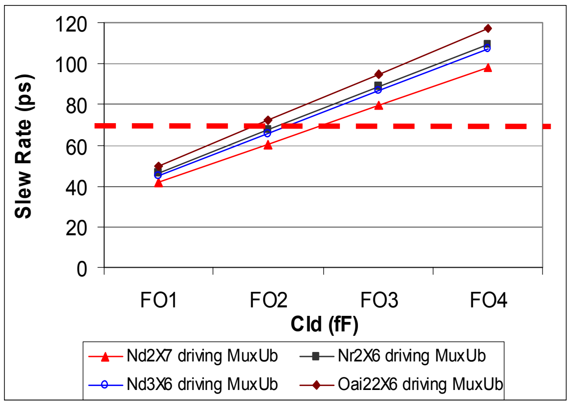

Figure 5. Slew of unbuffered Muxes.

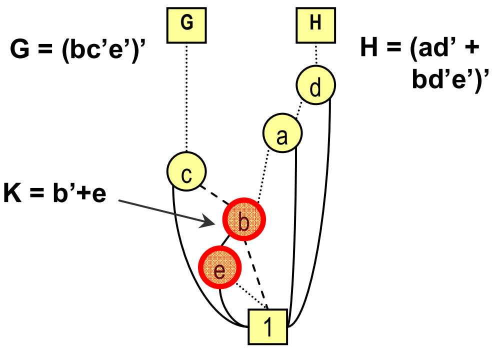

Figure 6. Common sub-factor detection.

V. Experimental Results

In order to isolate the brick generation step for an apple-to-apple comparison, we devised the following experiment. We generated a large number of random functions with support sizes in the range of 5 to 10, and implemented those functions using a commercial tree mapping approach (Design Compiler, DC) [13] and our brick generator, PBBG. The rationale behind this experiment is: if our brick generator performed consistently better than DC (which was used in the prototype design flow presented in [6]) over a large number of functions, then we are reasonably confident that our brick generator will improve the overall results when plugged into the overall Regular Logic Brick design methodology. We further propose a figure of merit to grade the quality of the bricks.

A. Figure of Merit

Since logic bricks are used as a technology library to implement IC designs, one way to grade their quality is to implement benchmark designs with various bricks libraries, and compare the area / delay / power of the resultant implementations. However, such comparisons would unavoidably incorporate other factors such as the contents of the library with respect to the requirements of the target designs, and the intricacies of the high level synthesis, which could obscure the effects of the brick generator.

Instead, we propose a figure of merit (FOM) based on logical effort [14], which is independent of the final implementation of IC designs. As shown in Figure 7 (a), if we plot the normalized delay (w.r.t. to the reference inverter) of a logic brick against the load it is driving in terms of fanout (i.e. Cout/Cin), the slope of the graph, g, is the logical effort of the brick, which is a measure of how good a driver the brick is compared with the reference inverter. The intercept on the delay-axis, p, is the parasitic delay of the brick.

The FOM we propose is the area under the graph over a relevant range of loads. The significance of this FOM is as follows: if we integrate the graph over a range of loads, and divide the integral by that range, we obtain the average normalized delay over the range of loads. Hence, the area under the graph is proportional to the average delay. We chose the typical load to be between fanout-of-one (FO1) and fanout-of-four (FO4), hence the range of Cout/Cin is [1,4] as shown in Figure 7 (b). The area under graph is easy to compute once p and g are known as it is simply the area of a trapezium.

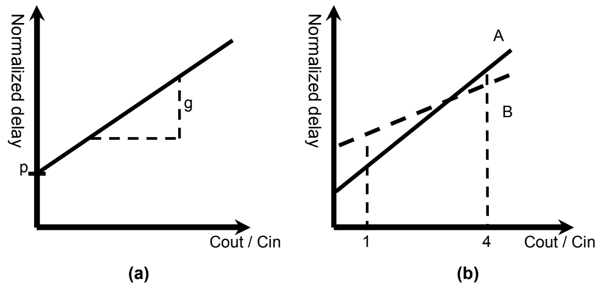

Figure 7. A logical effort based figure of merit.

Notice that the proposed FOM accurately reflects the fact that although brick A has a higher logical effort (slope) than brick B, its lower parasitic delay (intercept) makes it a better implementation than brick B (assuming both A and B perform the same logic function) over the range of interest because, on average, brick A will have lower delays than brick B. Of course, this argument is only valid if the load distribution is uniform over the range. Nevertheless, in the absence of other design specific information, this is a reasonable assumption.

B. Comparison with Design Compiler

Using the FOM described above we graded a collection of bricks generated for a commercial 65nm CMOS technology. The logical effort and parasitic delay of each brick is found via SPICE simulations similar to those outlined in Chapter 5 of [14]. Table 1 summarizes the results for this experiment. Each row of Table 1 reports the results for bricks with support size indicated in the left-most column. The columns with headings 'DC' and 'PBBG' are the measured FOM (lower is better) and area for bricks generated using DC and PBBG respectively. The numbers reported are the average over all bricks for each support size. On average, there is about 5% to 10% improvement in the FOM, with some of the larger improvement seen on bigger bricks as those are the ones more likely for PBBG to find some efficient Mux-decompositions and PT/static implementations that a more general synthesis approach might miss.

Table 1 Comparison with Design Compiler.

Support Size	FOM (average)		Area (average)	
	DC	PBBG	DC	PBBG
5	48.8	46.7	18.9	18.2
6	54.2	50.5	23.7	23.0
7	69.1	62.4	27.8	26.8
8	88.3	82.6	33.5	32.4
9	91.4	87.2	37.3	36.2
10	100.9	92.7	42.4	41.6

VI. Conclusions and Future Work

We have presented the benefits of implementing IC designs on regular fabrics using a small collection of logic bricks. We also presented an automated primitive-based logic brick generator, and proposed a figure of merit to grade the quality of logic bricks. Based on the proposed figure of merit, our experimental results demonstrated that logic bricks generated by our procedure are on average superior to those generated by a classical commercial synthesis approach. As a final remark, we would like to caution that we do not claim that our procedure is better than Design Compiler in general, as DC is undoubtedly a more comprehensive tool that is extremely versatile and scalable to handle huge designs and large technology libraries. Our only claim is that for the specific problem of logic bricks generation, a specialized tool, such as the one that we are proposing here, can provide substantial improvement. Future work includes integrating our brick generator with a Regular Fabrics design flow to further evaluate the merits of our brick generator.

REFERENCES

[1] T. Jhaveri, et al. "Maximization of layout printability / manufacturability by extreme layout regularity," SPIE 2006.

[2] A. J. Strojwas, "Process-Design Interaction: Modeling Based Design for Manufacturability," Tutorial, DAC, June 2003.

[3] M. Lavin, "Backend CAD flows for restrictive design rules," *Proc. of ICCAD*, Nov 2004, pp. 739-746.

[4] F. Mo, and R. Brayton, "PLA-based regular structures and their synthesis," *Trans. of CAD*, June 2003, pp. 723-729.

[5] Y. Ran, and M. Marek-Sadowska, "Designing a via-configurable regular fabric," *Proc. of CICC*, 2004. pp. 423-426.

[6] V. Kheterpal, et al. "Design methodology for IC manufacturability based on regular logic-bricks," *Proc. of DAC*, June 2005, pp. 353-358.

[7] A. Koorapaty, et al. "Heterogeneous programmable logic blocks," *Proc. of DATE*, Mar 2003, pp. 1118-1119.

[8] S. Yamashita, et al. "Pass-transistor / CMOS collaborated logic: the best of both worlds," *Symp. on VLSI Circuits*, June 1997.

[9] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[10] C. Yang, and M. Ciesielski, "BDS: a BDD-based logic optimization system," *Trans. on CAD*, July 2002, pp. 866-876.

[11] T. Stanion, and C. Sechen, "Boolean division and factorization using binary decision diagrams," *Trans. on CAD,* Sept. 1994, pp. 1179-1184.

[12] L. Hellerman, "A measure of computational work," *Trans. on Computers*, vol. c-21, May 1972.

[13] http://www.synopsys.com/products/logic/design_compiler.html

[14] I. Sutherland, R. Sproull, and D. Harris, *Logical Effort.* Academic Press, 1999.