TRIPLE-THRESHOLD STATIC POWER MINIMIZATION TECHNIQUE IN HIGH-LEVEL SYNTHESIS USING 90NM MTCMOS TECHNOLOGY

by

Harry I-An Chen
B.A.Sc. (First Class Honors), Simon Fraser University, 2005

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

In the
School
of
Engineering Science

SIMON FRASER UNIVERSITY

Summer 2007

APPROVAL

Name: Harry I-An Chen

Degree: Master of Applied Science

Title of Thesis: Triple-Threshold Static Power Minimization Technique in High-Level Synthesis Using 90nm MTCMOS Technology

Examining Committee:

Chair: **Dr. Ash Parameswaran**
Professor of Engineering Science

Dr. Marek Syrzycki
Co-Senior Supervisor
Professor of Engineering Science

Dr. James Kuo
Co-Senior Supervisor
Adjunct Professor of Engineering Science

Dr. Rick Hobson
Examiner
Professor of Engineering Science

Date Defended/Approved: July 18, 2007

ABSTRACT

As CMOS System-on-Chips approach the limits of power dissipation, static power has become dominant in a circuit's total power dissipation. The static power is increasing exponentially as technology nodes shrink and is projected to exceed the dynamic power within the near future. Techniques that use the multi-threshold CMOS (MTCMOS) technology have been developed to reduce static power effectively. In this thesis, a novel triple-threshold static power minimization technique in high-level synthesis has been developed using the 90nm MTCMOS technology. Using static timing analysis, the optimal partitioning of gates with three different threshold voltages is determined via iterative analysis. The proposed triple-threshold technique has been applied to optimize several benchmark circuits, and the results show an average saving in static power close to 90% compared to un-optimized LVT designs. For all designs tested, the triple-threshold technique has produced designs with lower static power compared to a dual-threshold technique.

Keywords: multi-threshold; triple-threshold; static power reduction; low power; high-level synthesis; digital CMOS VLSI

Subject Terms: Electric Leakage Prevention; Metal oxide semiconductors, Complementary -- Design and construction; Integrated circuits -- Design and construction; Digital integrated circuits

To my family...

ACKNOWLEDGEMENTS

I would like to thank Dr. Kuo for his direction, inspiration, and dedicated effort behind this thesis work. In the short time of a year and a half, you have taught me a great deal of information with your profound knowledge that I would not have learned elsewhere. Thank you for your kindness, your criticism, and your encouragements.

I would like to thank Dr. Syrzycki for his kind offer to continue to provide guidance through the rest of my graduate studies. Thank you for your dedication and helpful advices, and thank you for taking every effort to ensure my success. I am very grateful of your guidance throughout my undergraduate and graduate studies. Also, thank you for your kind supplies for making the lab a nice place for research.

I would like to thank Dr. Hobson for introducing the world of VLSI to me during my undergraduate studies. Without your educational course, this thesis work would not have been possible.

I would also like to thank Benjamin Chung at PMC Sierra. Thank you for providing your original TCL scripts and your help in jumpstarting my research.

Finally, I would like to give special thanks to my family and friends for the strong support and cheer during my graduate studies. Special thanks to my lab mates, Edward and Henry, for the memorable time spent together.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

BFS	Breadth-First Search
CAD	Computer-Aided Design
CMC	Canadian Microelectronics Corporation
CMOS	Complementary Metal-Oxide-Semiconductor technology
HVT	High Threshold Voltage
ICs	Integrated Circuits
ILP	Integer Linear Programming
ITRS	International Technology Roadmap for Semiconductors
LBT	Levelized Back-Tracing
LVT	Low Threshold Voltage
MISA	Maximum Independent-set-based Slack Assignment
MTCMOS	Multi-Threshold CMOS technology
NP	Non-deterministic Polynomial-time, a set of decision problems that can be solved in polynomial time in non-deterministic computer model
NP-Hard	Non-deterministic Polynomial-time Hard, a class of problems that is at least as hard as a problem in the NP set
RTL	Register Transfer Language
SDF	Specific Delay Fictitious-buffers
STA	Static Timing Analysis
SVT	Standard Threshold Voltage

TCL	Tool Command Language
TTL	Transistor-Transistor Logic, a standard for designing integrated circuits with a 5V power supply
VLSI	Very-Large-Scale Integration

1 INTRODUCTION

1.1 CMOS Development Trends

The CMOS technology development has been progressing steadily and rapidly over the past few decades. Transistor sizes have been scaled down at an exponential rate (Figure 1-1), allowing designers to integrate more transistors onto a chip. As the transistor density increases, the power density also increases. Designers no longer have to achieve just the simple goals of optimizing for speed and area, but to strive for a balance between speed, area and power dissipation.

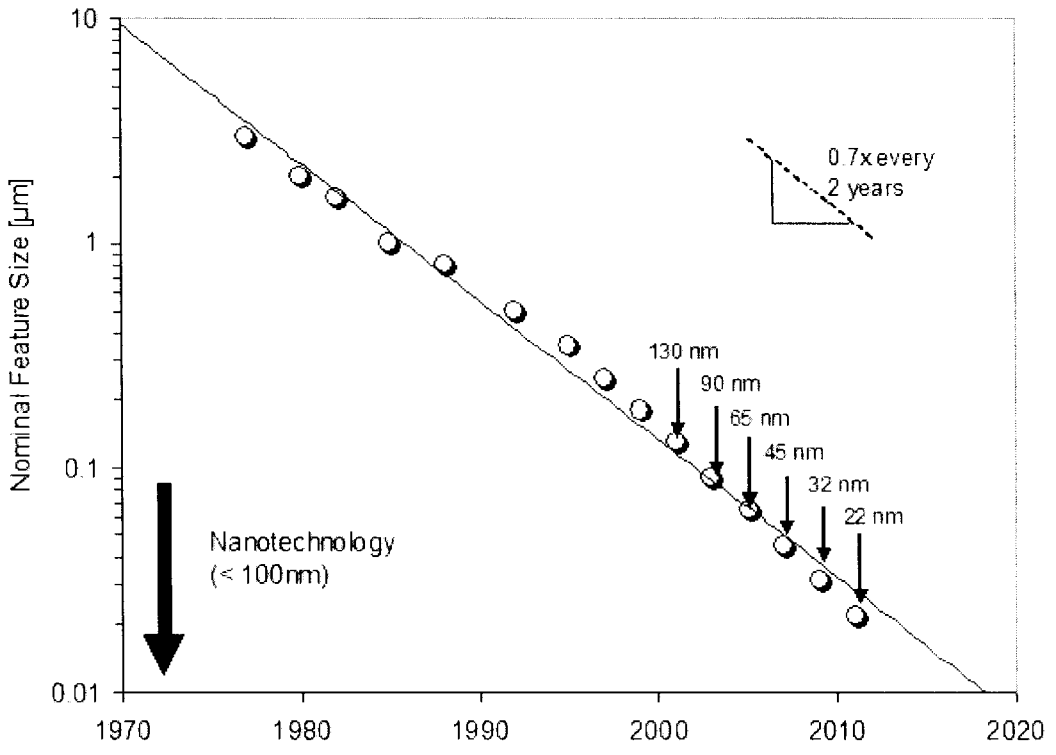

Figure 1-1. Scaling down of transistor sizes in CMOS development [1]

Traditionally, integrated circuits (IC) have been driven by high supply voltages, e.g. 5V for standard transistor-transistor logic (TTL). The use of standardized high supply voltages ensures compatibility between devices and provides large load-driving capabilities. However, as transistor sizes are scaled down, the load capacitance for each transistor in an IC decreases, and high load-driving capabilities are no longer necessary. Lowering the supply voltage to reduce the power dissipation becomes a viable option for designers.

Another factor leading to supply voltage reductions is associated with increasingly high internal electric fields in CMOS transistors. Reducing the channel length of a transistor while keeping constant the voltage drops over the gate oxide and the channel increases the electric field across the channel, which causes long-term reliability concerns due to impact ionization and hot carrier effects [2]. Reducing the supply voltage to relieve stress from high internal electric fields becomes necessary.

The total power dissipation of a CMOS circuit consists of dynamic power and static power. Dynamic power is the power dissipated due to a change in the input; static power is the power dissipated when the circuit is inactive. Typically, the main component of power dissipation in a CMOS circuit is the dynamic power, which can be estimated using Eq. 1.1 [3]:

$$P_{dynamic} = \alpha C_{load} V_{dd}^2 f_{clk} + t_{sc} V_{dd} I_{sc} f_{clk}, \qquad (1.1)$$

where α is the circuit's switching activity, C_{load} is the load capacitance, V_{dd} is the supply voltage, f_{clk} is the clock frequency, t_{sc} is the short-circuit time per clock cycle, and I_{sc} is the short-circuit current. The first term in the equation describes the switching power; the

second term describes the short-circuit power. Because of the quadratic relationship between dynamic power and the supply voltage, a linear reduction in the supply voltage results in a quadratic reduction in the dynamic power dissipation.

Although reducing the supply voltage results in large reductions in the power dissipation, the circuit's switching speed is compromised. For an NMOS transistor operating in the saturation region, the drain current can be expressed as [4]:

$$I_D = \frac{W\mu_n C_{ox}}{2L}(V_{GS} - V_T)^2 . \qquad (1.2)$$

As the supply voltage drops, V_{GS} decreases and the drain current decreases in a quadratic fashion, resulting in lower switching speeds. To counter this side effect, designers lower the threshold voltage to increase the speed. Figure 1-2 shows the trend in scaling the supply voltage and threshold voltage as projected by the International Technology Roadmap for Semiconductors (ITRS) in 2001.

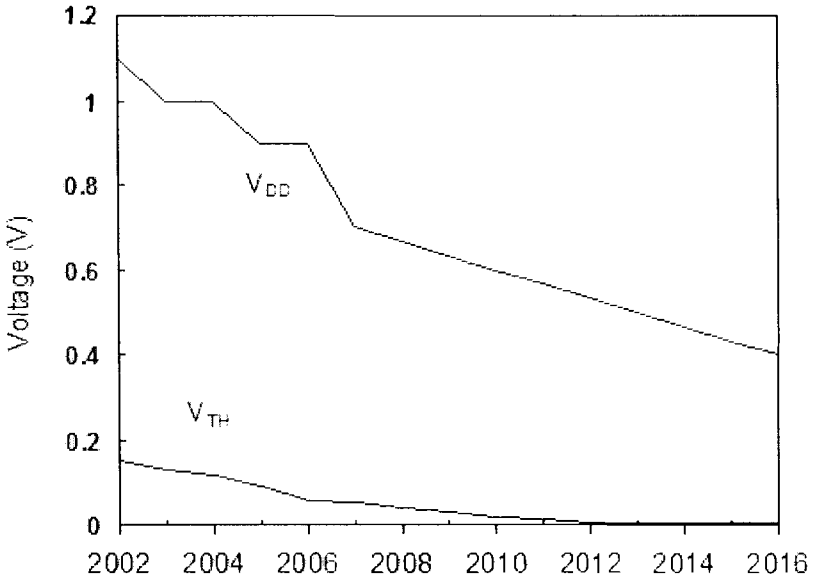

Figure 1-2. Projected supply voltage and threshold voltage

Lowering the threshold voltage introduces new problems as CMOS technology progresses towards the nanometre regime. For an MOS transistor operating in weak inversion mode, the minority carriers diffuse across the channel, resulting in leakage currents. Ideally when the gate potential is zero, the transistors should be cut off and the off-current should be zero. In practice, the subthreshold current depends exponentially on the gate potential and is non-zero at a zero gate potential [2]:

$$I_D \propto e^{\frac{V_{GS}}{nkT/q}}. \tag{1.3}$$

Plotting the drain current in logarithmic scale against the gate voltage produces a linear plot, and the slope of this line is known as the "subthreshold slope." Adjusting the threshold voltage of an MOS transistor shifts the plot horizontally, thereby shifting the y-intercept (representing the off-current) vertically. But because of the exponential scale on the vertical axis, the magnitude of the off-current changes exponentially.

Figure 1-3 shows the drain current simulation in HSPICE of three 90nm minimum-size transistors with different threshold voltages. The subthreshold leakage current at $V_{GS} = 0$ is increased by an order of magnitude when the threshold voltage is decreased by 0.08V. The exponential increase in the leakage current is undesirable and results in the exponential increase in the static power of the circuit.

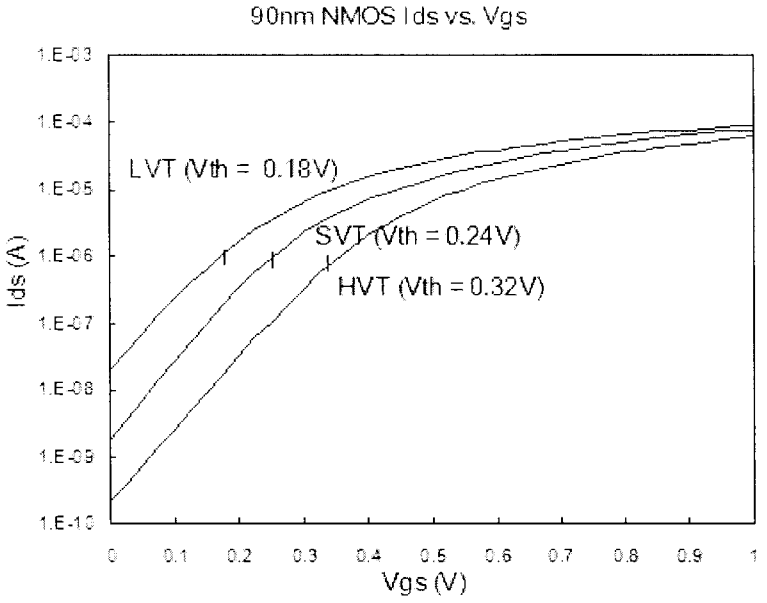

Figure 1-3. Subthreshold current for transistors with different threshold voltages

As CMOS technology development continues, the static power becomes increasingly more dominant in the total power dissipation envelope. Figure 1-4 shows the power trend for the past few decades as reported by Intel [5].

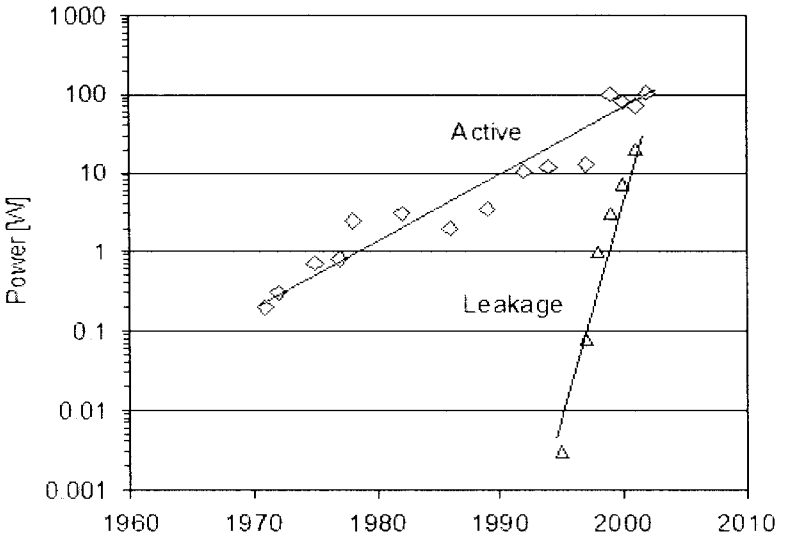

Figure 1-4. Power trend in the past few decades of CMOS development [5]

Traditionally, the dynamic power has been the dominant factor in the total power, and the static power has always been negligible. In recent years, however, the static power has become increasingly more dominant. If the current trend continues, the static power is predicted to contribute to most of the power in a CMOS very-large-scale integration (VLSI) system, even surpassing the dynamic power. Figure 1-5 shows the projected power dissipation per gate in future CMOS development [6]. The dynamic power dissipation per gate can be reduced by lowering the supply voltage. Nevertheless, the leakage power per gate continues to increase as the threshold voltage is reduced.

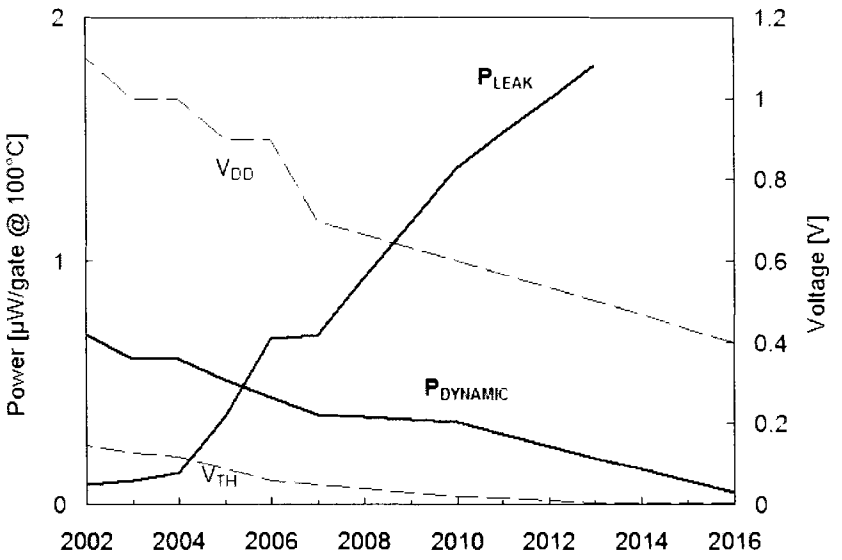

Figure 1-5. Predicted power trend in future CMOS development [6]

Faced with the new problem of having increased static power dissipations in a circuit, designers must investigate ways to reduce the static power. Some areas of research currently being pursued are:

- Research new materials for low-leakage transistors.

- Design transistors with variable body-bias for dynamic threshold-voltage adjustments.
- Utilize multi-threshold CMOS (MTCMOS) technology.

The MTCMOS technology utilizes transistors with different threshold voltages to reduce static power. Combining the strengths of slow and less leaky high threshold-voltage (HVT) transistors with fast but leaky low threshold-voltage (LVT) transistors, the MTCMOS approach may be the most cost-effective technique. Manufacturers of the nanometre CMOS technology usually provide transistor models with a predetermined set of threshold voltages. HVT transistors can be placed in designs where timing requirements are less stringent, thus saving static power; transistors with the standard threshold voltage (SVT) can be used for typical applications; and LVT transistors can be used for high-speed applications at the expense of dissipating more static power. To fully exploit the MTCMOS technology, it is possible to reduce the static power by placing HVT transistors in slower timing paths while placing LVT transistors in high-speed paths.

Several different categories of MTCMOS techniques exist in the literature, and the results have been promising. The MTCMOS approach can use existing technology as well as preserve the high-level circuit designs, making it a practical and efficient approach for reducing static power. This research project is focused on the MTCMOS approach, and a new triple-threshold methodology for static power minimization is presented.

1.2 Research Goals

In the current highly competitive IC industry, designing for low-cost, low-power and high-speed applications with the shortest time-to-market is important. Given these criteria for IC designs, the MTCMOS approach may be the most cost-effective way to minimize the static power dissipation of a circuit. To ensure fast time-to-market, using MTCMOS techniques in high-level synthesis is ideal. Numerous dual-threshold MTCMOS techniques have been published in literature. However, publications on using the triple-threshold technology are scarce. To expand the research in this area, this research focuses on developing a high-level triple-threshold optimization technique.

The research goals are as follows:

- To investigate current MTCMOS techniques.
- To develop a 16-bit multiplier circuit as a test vehicle for analyzing the static power dissipation.
- To propose a new methodology using the 90nm triple-threshold technology.
- To evaluate the effectiveness of the new methodology and compare with current methodologies using the 16-bit multiplier as well as other benchmark circuits if available.

The proposed technique should also meet the following criteria:

- The technique should be applicable for optimizing existing as well as future gate-level netlists.

- It should provide savings in the static power dissipation that are comparable or better than current techniques.
- The circuit's operating clock speed should not be compromised while the static power is minimized.
- The circuit's area should not be significantly increased.
- The optimization run-time should be reasonable and comparable to existing techniques.

The final deliverable is a high-level triple-threshold static power reduction methodology that is suitable for designing low-power high-speed digital CMOS VLSI designs.

1.3 Thesis Outline

This thesis is organized as follows. Chapter 2 discusses prior work on MTCMOS static power reduction techniques. Different approaches and algorithms will be presented and analyzed. Chapter 3 describes the simulation tools used in the experiments. Chapter 4 describes the proposed triple-threshold technique, and the simulation results are presented and analyzed in Chapter 5. Chapter 6 concludes this thesis.

2 RELATED WORK ON MTCMOS TECHNIQUES

This chapter presents an overview of MTCMOS circuit design techniques for reducing static power. Benefits and drawbacks for each technique will be presented. This work utilizes the MTCMOS gate partitioning technique, and thus gate partitioning algorithms will be described in more detail. Most gate partitioning algorithms utilize the dual-threshold technology. Nevertheless, the relative efficiency and effectiveness of each algorithm can be applied to triple-threshold algorithms.

2.1 MTCMOS Technique Overview

The MTCMOS technique relies on the use of transistors with different threshold voltages to reduce the leakage current in a circuit. HVT transistors have low subthreshold leakage currents and slow switching speeds; LVT transistors have fast switching speeds and high leakage currents. To combine the strengths of both HVT and LVT transistors in an MTCMOS circuit, several circuit design techniques have been developed:

1. **Source/body biasing:** A bias voltage is applied to the source or body of an "off" device to increase the threshold voltage due to body effect. The subthreshold leakage current is reduced as a result.
2. **Virtual supply rails:** LVT circuits are isolated from the power and ground rails by placing HVT sleep transistors in series with V_{DD} and/or ground to provide virtual supply rails.

3. **Gate Partitioning:** LVT gates are placed in timing-critical paths and HVT gates are placed in non-critical paths to reduce the overall static power dissipation.

2.1.1 Source/Body Biasing

The source or body biasing technique uses a biasing voltage for the source or substrate when a transistor is in the "off" state. For an NMOS transistor, the threshold voltage can be expressed as follows [2]:

$$V_T = V_{T0} + \gamma\left(\sqrt{V_{SB} - 2\phi_f} - \sqrt{-2\phi_f}\right). \quad (2.1)$$

When a positive biasing voltage V_{SB} is applied, the threshold voltage is increased.

The biasing voltage V_{SB} can be controlled by biasing the body potential [19][20] or by biasing the source terminal [21][22], as shown in Figure 2-1. During normal operations, V_{SB} is kept at zero to maintain the nominal threshold voltage. In sleep mode, V_{SB} is increased, which increases the threshold voltage and lowers the subthreshold leakage current. A side effect, though, is an increase in the reverse PN-junction leakage current from the source to substrate.

The biasing techniques require extra circuitry to implement the biasing voltage. As a result, the circuit area is larger, especially for body biasing since each transistor requires a separate well. The biasing techniques also require custom layouts for each gate, which increases the design cost and complexity.

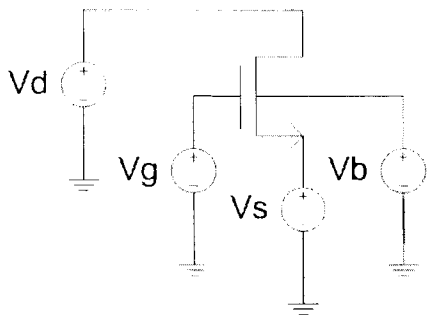

Figure 2-1. Reducing subthreshold leakage current by adjusting V_s or V_b

2.1.2 Virtual Supply Rail

For mobile applications with long idle times, devices can be put into sleep mode to reduce power. The power supply and ground rails are isolated with HVT sleep transistors to provide virtual supply rails for the logic blocks, as shown in Figure 2-2. Although the figure shows that both the power and ground rails are gated with sleep transistors, only one polarity sleep transistor is required if the logic block is purely combinational [8]. During normal operations, the sleep transistors are turned on and the logic blocks can operate at fast speeds with the LVT transistors. In standby mode, the supply rails are switched off and the subthreshold leakage currents are reduced with the use of the HVT sleep transistors [7][9][10][11].

Implementation of virtual supply rails can vary. The most basic design method is to add a sleep transistor for each logic gate. The standard cell libraries provided by manufacturers cannot be used with this method, and custom designs for each gate is required, which costs considerable design time. The area penalty is very large due to the large number of sleep transistors being used, and extra complexity is introduced to the routing and buffering of the global sleep signal.

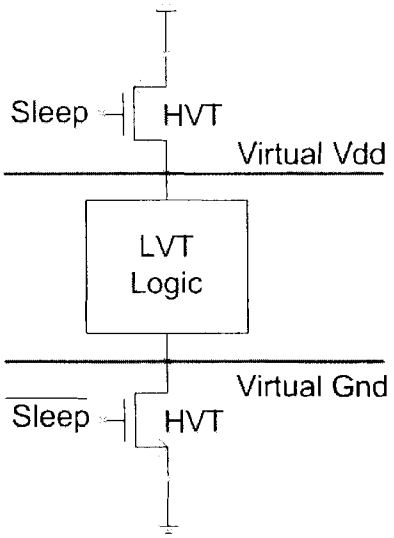

Figure 2-2. Virtual power/ground rails isolated by HVT sleep transistors

Instead of using a sleep transistor for every logic gate, the other extreme design method is to use one very large sleep transistor to provide a virtual rail for a large block of gates. Standard cell libraries can be used to implement the logic block, and the area penalty is minimal. However, due to the large parasitic capacitances in the virtual rails, the switching speed is penalized when changing between the sleep and active modes. The sleep transistor may also be larger than necessary, since not all transistors may be switching at the same time.

More fine-grained approaches have been proposed to cluster gates together based on discharge current patterns [12][13][14][15]. Gates that have mutually-exclusive discharge current patterns can share the same sleep transistor without having to increase the sleep transistor size; gates with discharge currents that do not exceed a set limit can also be clustered. The fine-grained algorithms for transistor sizing require more optimization run time. With the advance of computer technology, however, the design cost diminishes.

Regardless of the implementation, an inherent problem for virtual supply rails is the inability to retain state information during sleep mode. Special sequential circuits are required for state retention [16]. For some circuits where MTCMOS gates are connected to standard CMOS gates, sneak leakage paths may also exist. Extra effort is necessary to eliminate the leakage paths [16][17][18].

The virtual supply rail technique is a common technique to reduce power in mobile applications during standby mode. However, the technique cannot be used to reduce leakage power during the active mode. In fact, the power dissipation may increase slightly during the active mode because of the extra sleep transistors. For applications that are active most of the time, the virtual supply rail technique is ineffective in reducing static power.

2.1.3 Dual-Threshold Transistor/Gate Partitioning

The dual-threshold partitioning techniques place LVT devices in timing-critical paths and HVT devices in non timing-critical paths. Since the subthreshold leakage current depends exponentially on the threshold voltage, changing transistors from LVT to HVT can result in leakage current reductions by a few orders of magnitude [8].

To partition a circuit into HVT and LVT transistors, the circuit's critical paths must be identified. For optimal partitioning, a transistor in the non-critical path should only be replaced by a transistor of higher threshold voltage if the leakage power reduction is maximal and the delay penalty is minimal. Identifying which transistors or gates to be replaced is known to be a non-deterministic polynomial-time hard (NP-hard) problem [23][24], i.e. the complexity is at least as hard as a polynomial-time problem. In the

worst-case scenario, solving the problem requires run times that depend exponentially on the circuit size. To achieve acceptable run times, heuristic algorithms have been developed for near-optimal partitioning. More details on these algorithms will be covered in the following sections.

A drawback of the dual-threshold partitioning technique is the requirement of extra masks during the manufacturing process. Also, for designs with many critical paths, partitioning may be ineffective. However, the partitioning technique can reduce leakage currents during both active and sleep modes, and standard cell libraries can be used. Compared to the biasing or virtual supply rail techniques, gate partitioning requires less circuit design time, does not require extra circuitry, and can provide leakage power reductions at all times. Gate partitioning is therefore a more attractive method for reducing leakage power for general applications. For mobile applications, partitioning may also be used in conjunction with the virtual rail technique to further reduce power in sleep mode [30][31].

2.2 Dual-Threshold Partitioning Algorithms

The dual-threshold algorithms can be separated into two broad categories. The LVT to HVT algorithms start with an LVT circuit and selectively replace gates in the non-critical paths from LVT to HVT. The HVT to LVT algorithms start with an HVT circuit and selectively replace gates in the critical paths from HVT to LVT. Depending on the circuits and the availability of computer-aided design (CAD) tools, both types of algorithms may be similarly effective in reducing static power.

2.2.1 LVT to HVT Algorithms

The LVT to HVT algorithms initialize a design with LVT gates. LVT gates in the non-critical paths are replaced by HVT gates based on different criteria. Some proposed algorithms are: breadth-first search (BFS) [25], levelized back-tracing [26], levelized maximum cut [27], maximum independent set [28][29], and solving specific delay fictitious-buffers (SDF) as an integer linear programming (ILP) problem [24].

2.2.1.1 Breadth-First Search

The BFS algorithm [25] traces gates backwards from the primary outputs. The delay time of each gate is calculated and recorded. The maximum amount of time that a gate's delay can be increased without affecting the circuit's overall performance is recorded as the slack time. During the back-tracing search, if the slack of a gate is positive and changing the gate's threshold to HVT does not result in a negative slack, the gate is changed to HVT. The algorithm continues the back-tracing search for one primary output until all gates have been back-traced, and the algorithm continues to back-trace from the next primary output until all primary outputs have been back-traced.

The BFS algorithm is a fast algorithm but may not result in the most optimal designs, since the replacement of gates does not depend on the weight of the delay or power, but rather on the order in which gates are visited. An algorithm that prioritizes the replacement of gates based on a weight of the amount of power saved vs. the time delay increase may be more effective in partitioning the gates. Also, back-tracing from a fixed primary output may affect the path slacks for other primary outputs. A levelized approach where gates in the same level are replaced before gates in the next level are replaced may produce better results.

2.2.1.2 Levelized Search

A levelized search algorithm assigns a level number to every gate in the circuit. The algorithm starts from assigning a level number zero to each primary input. Then, all gates are assigned a level number that is one greater than the maximum level number of the immediate fan-in gates. The levelized back-tracing (LBT) algorithm [26] is similar to the BFS algorithm, except that gates in the same level are replaced before gates in the next traversed level are replaced. The algorithm traces from the maximum level back towards the primary inputs. For all gates in each level, if replacing a gate with a high threshold does not result in a negative slack, the gate is assigned a high threshold voltage. The back-tracing continues until level zero is reached.

The LBT algorithm improves upon the BFS algorithm by prioritizing the assignment of gates in the same level to HVT. Results of optimizing benchmark circuits show that the LBT algorithm produces circuits with lower leakage power than BFS for all the circuits tested [26]. The LBT algorithm, however, still does not assign weights to the gates or levels, and thus more power reductions may be possible with a weighted search algorithm.

2.2.1.3 Maximum Cut

A gate-level circuit can be represented as a directed acyclic graph, where each gate is represented as a node, and each fan-in or fan-out connection is represented as a directed edge. A cut of a directed acyclic graph is a partition of the nodes into two disjoint sets. The levelized maximum cut algorithm in [27] solves the gate partitioning problem by iteratively finding cuts with the maximum weight in a circuit. The weight is defined as the power saving of each node by replacing the LVT gate with an HVT gate.

To reduce the complexity for finding all possible cuts in the graph, the levelized approach assigns a level number for each gate, and cuts are only made at the boundary between each level (Figure 2-3). Gates in the level with the maximum total weight are selected for replacement in each iteration process. The algorithm continues until no more levels can be selected without causing a negative slack.

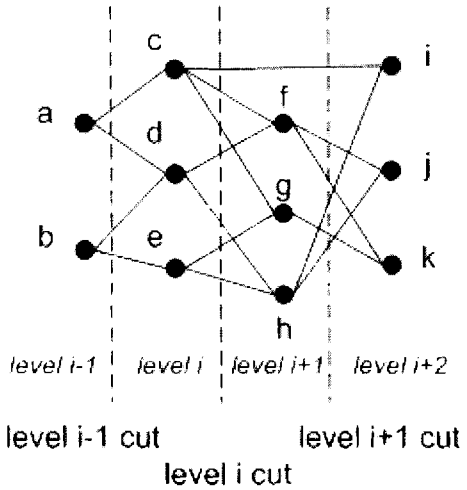

Figure 2-3. Levelized maximum cut for a circuit represented as an acyclic graph [27]

The levelized maximum cut algorithm replaces gates with larger reductions in static power first. Compared to BFS and LBT, the levelized maximum cut algorithm produces designs with lower static power.

2.2.1.4 Maximum Independent Set

The maximum independent set algorithms solve the gate partitioning problem by finding the maximum set of gates that can be changed to HVT without degrading performance.

The maximum independent-set-based slack assignment (MISA) algorithm in [28] weighs each node based on the effective power saving, the change in the slack, and the

likelihood that an adjacent node is also an HVT node (for manufacturing yield considerations). All nodes that may be changed to HVT without causing a negative slack are placed in a candidate list. The greedy algorithm, i.e. an algorithm that selects the locally optimum choices at each stage with the hope of finding the globally optimum solution, is used to iteratively select the node with the maximum weight and determine if the node can be changed to HVT without causing a negative slack.

The algorithm in [29] is similar to [28], but weighs nodes based on the effective power saving only. The greedy approach is also used to iteratively select and replace the nodes with the maximum weight while keeping the slack non-negative.

The MISA algorithm in [28] is the most computationally intensive approach. For large applications, the efficiency may be lower compared to algorithms in [27] and [29]. The algorithm also considers placing HVT nodes adjacent to each other to improve yield, which may result in suboptimal placement for reducing static power. However, since slack is also being weighted and gates with smaller impacts on the slack are replaced first, the algorithm in [28] may potentially be able to replace more LVT gates to HVT than the algorithm in [27] or [29].

Algorithms in [27], [28] and [29] are all greedy algorithms. The levelized approach in [27] is not as fine-grained as in [29]. However, as with all greedy algorithms, unique worst-case scenarios may exist where the greedy algorithms are suboptimal [32]. Hence, in some cases the algorithm in [27] may produce better results than the algorithm in [29], and vice versa.

2.2.1.5 Integer Linear Programming

The ILP algorithm in [24] inserts buffers into a circuit for delay balancing. The delay buffers are fictitious entities that are used solely for the purpose of modelling the slack in a circuit. The selection of gates is solved as an ILP problem to minimize the total static power. After selecting and replacing the gates, the threshold voltage for the HVT gates that results in the lowest static power is determined.

Compared to other LVT to HVT algorithms, the ILP algorithm generates designs with the lowest static power. However, the ILP algorithm is also significantly more computationally intensive, and thus it is unsuitable for large designs.

2.2.2 HVT to LVT Algorithms

The HVT to LVT algorithms initialize a design with HVT gates. An HVT circuit is slower than an LVT circuit, and thus the slow HVT gates in critical paths need to be replaced with fast LVT gates to improve the performance. Algorithms that have been proposed for HVT to LVT replacement are: breadth-first search (BFS) [33], minimum cut [34], maximum cut [34], and maximum independent set [35][36][37].

2.2.2.1 Breadth-First Search

The BFS algorithm in [33] is similar to the BFS algorithm in [25], except that weights have been assigned to the gates to improve the optimization process. The algorithm traverses backwards from the primary outputs, and in each iteration run, the gate with the maximum timing delay reduction is changed to LVT. Timing information is updated and new critical paths are selected after each gate replacement. This algorithm

combines BFS with the maximum independent set approach, and the results have been shown to be better than algorithms in [24] and [25].

2.2.2.2 Minimum Cut

The minimum cut algorithm in [34] assigns weights to each node based on the increase in the static power and the reduction in the time delay when a node is changed from HVT to LVT. The algorithm searches for cuts that have the minimum total weight in each iteration process, i.e. cuts where gates can be changed to LVT with the least increase in the static power and the most time delay reduction. Gates that are not in the critical path are assigned a weight of infinity.

Finding the minimum cut of a weighted graph has been studied extensively. The well-known solution to the minimum cut problem is the max-flow-min-cut algorithm [38]. The fastest algorithm is the preflow-push algorithm [39] and is the algorithm implemented in [34].

In [34], the minimum cut algorithm is shown to be less effective than the maximum cut algorithm in [27] (presented in Section 2.2.1.3). The main reason is that the assignment of infinite weights to the gates in non-critical paths misleads the algorithm to a suboptimal solution [34]. The maximum cut algorithms do not require the assignment of infinite weights and thus are more effective than the minimum cut algorithm.

2.2.2.3 Maximum Cut II

The maximum cut II algorithm in [34] initializes a design with HVT gates, but all gates in the critical paths are replaced with LVT gates. For the subset of LVT gates in

the circuit, the maximum cut algorithm in [27] is used to search for and replace gates back to HVT, as shown in Figure 2-4.

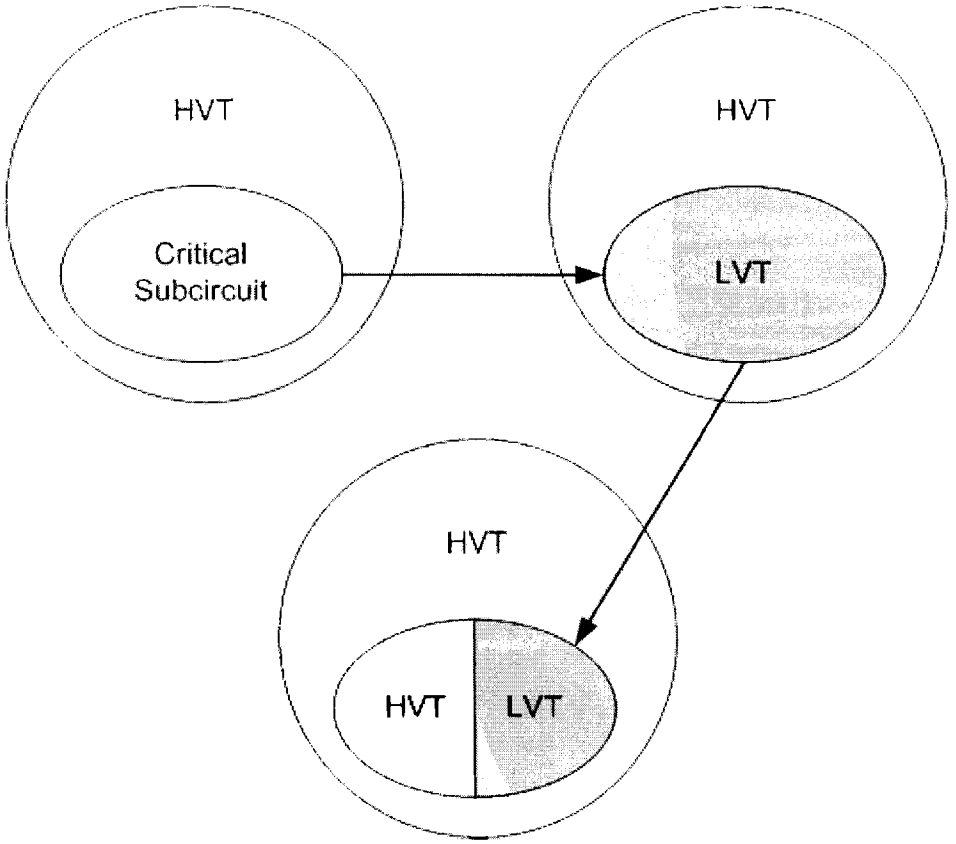

Figure 2-4. The maximum cut II algorithm flow diagram [34]

Both the maximum cut and maximum cut II algorithms have similar results in static power reductions and are shown to be more effective than the minimum cut algorithm [34]. However, the maximum cut II algorithm only needs to process a smaller subset of the circuit and is therefore a more efficient algorithm than the maximum cut algorithm in [27].

2.2.2.4 Maximum Independent Set

The maximum independent set algorithms in [35] and [36] initialize designs using HVT gates. Each gate is assigned a weight, and in each iteration run the gate with the maximum weight is selected and replaced by an LVT gate to reduce the timing delay. In [35], the weight of a gate is defined as the maximum delay time reduction when the gate is changed from HVT to LVT. In [36] and [37], the weight of a gate is defined as the number of critical paths passing through the gate.

The algorithm in [35] replaces gates with the maximum time delay reduction. However, the gate being replaced may be in a critical path where only a slight decrease in the time delay is necessary to meet the timing constraint. Replacing a gate with the maximum time delay reduction when unnecessary may result in larger static power, and therefore the algorithm in [35] may be suboptimal.

The algorithm in [36] addresses this problem by selecting gates where most critical paths pass through. By replacing these bottleneck gates first with LVT gates, the time delays for the most number of critical paths can be reduced at once. This algorithm is therefore potentially more effective than other weighted algorithms. It will be used for comparison purposes to evaluate the performances of dual-threshold and triple-threshold algorithms.

2.2.3 Transistor-Level vs. Gate-Level

Transistor-level circuits are typically modelled with tools such as HSPICE, which provides accurate simulations on the transistor currents and timing delays. In gate-level modelling, each gate is characterized with a set of parameters for the area, timing and power. In general, simulations at the transistor-level are computationally more intensive

than gate-level simulations, and for large designs, gate-level simulations are more efficient.

A drawback of gate-level simulations is that transistors within a gate must be either all LVT or all HVT. Since not all inputs of a gate may be part of a critical path, the transistors connected to inputs in the non-critical path can be assigned to HVT while transistors connected to inputs in the critical path can be assigned to LVT. Algorithms for transistor-level MTCMOS assignments have been proposed [40][41][42]. Nevertheless, given the trend of a growing number of gates being integrated into a chip, these transistor-level optimization techniques may be unsuitable for current and future designs.

2.3 Prior Triple-Threshold Techniques

The triple-threshold techniques utilize three different threshold voltages on a chip to provide more fine-grained control on transistor leakage currents. Prior work using the triple-threshold technology has been very limited – the triple-threshold technology has just become a possibility in the 90nm CMOS technology node. A triple-threshold technique that combines the use of the virtual rail technique and the partitioning technique has been proposed, as shown in Figure 2-5 [30][31].

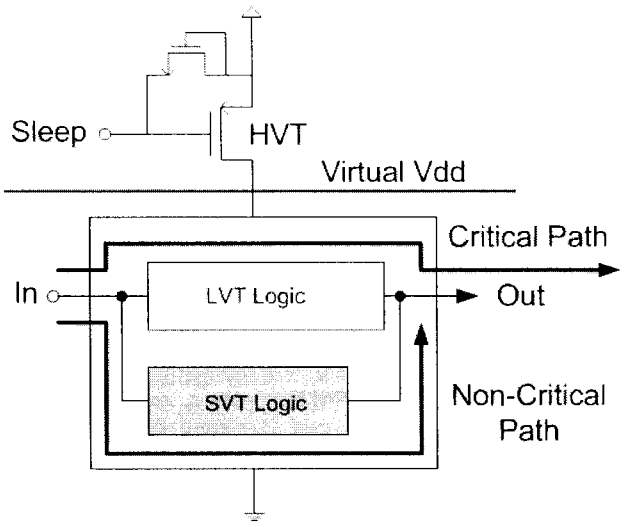

Figure 2-5. Triple-threshold technique that combines the virtual rail technique and the dual-threshold gate partitioning technique [30][31]

The prior triple-threshold technique uses HVT transistors to implement the sleep function. In the logic blocks, the threshold voltages are assigned to gates using a dual-threshold technique. The manufacturing technology requires two more masks than conventional CMOS manufacturing. The ion implantation for HVT transistors is implemented using both masks for the SVT and LVT implantation, and the ion implantation concentration for HVT transistors is the sum of the ion implantation concentrations for SVT and LVT.

The technique combines the benefits of the virtual rail technique and the gate partitioning technique. The static power has been reduced during both standby and active modes while not compromising the circuit speed. However, SVT and LVT gate have been used inside logic blocks for speed considerations. Because SVT transistors have larger subthreshold leakage currents than HVT transistors, the static power may be higher compared to HVT+LVT dual-threshold circuits during the active mode. Further static

power reduction may be possible with a more fine-grained approach by using HVT gates inside the logic blocks in addition to the SVT and LVT gates.

2.4 Summary

This chapter has presented an overview on MTCMOS techniques for static power reduction. The biasing, virtual supply rail and gate partitioning techniques have been presented and compared. The gate partitioning technique is a better approach in terms of design cost and silicon area, and algorithms for dual-threshold partitioning have been presented. Prior triple-threshold techniques have also been presented in this chapter. Previous publications using the triple-threshold technology have been scarce. The only prior work found has not fully exploited the capabilities of the triple-threshold technology, and more fine-grained optimization may be possible. Based on prior work information, more static power reductions may be achieved with a fine-grained triple-threshold gate partitioning technique.

3 SIMULATION TOOLS

This chapter describes the simulation tools used for the experiments. The tools chosen for this thesis are Synopsys Design Compiler™ and PrimeTime™, both made available by the Canadian Microelectronics Corporation (CMC). Synopsys Design Compiler™ is a convenient tool for high-level synthesis, while PrimeTime™ provides accurate static timing analysis and extraction of critical paths. Both tools include the support of scripting using the tool command language (TCL), which enables fast development and testing of optimization algorithms.

3.1 Support for TCL Scripting in Synopsys Tools

A major advantage of using Synopsys tools is the integrated support for TCL scripts. The TCL language provides basic programming constructs such as variables, loops, and procedures. Scripts can be written to process return values from Synopsys commands and iteratively perform optimization steps. TCL is a scripting language and thus scripts do not need to be compiled into a machine language before execution, which allows for rapid script development and debugging.

Synopsys Design Compiler™ supports two modes of operation. The legacy mode can be started using the *dc_shell* command, while TCL mode is started with the *dc_shell-t* command. PrimeTime™ only runs in one mode with integrated TCL support. To invoke PrimeTime™, the *pt_shell* command is used.

3.2 Synopsys Design Compiler™

Design Compiler™ is a tool for fast synthesis of register transfer level (RTL) circuits. In this work, Design Compiler™ is used for circuit synthesis, power dissipation estimation, and cell usage report.

3.2.1 Circuit Synthesis

A typical synthesis flow consists of the following steps:

1. Read in a high-level or gate-level RTL design written in the VHDL or verilog language.
2. Set a timing constraint for the circuit to specify a target clock period.
3. Load logical and physical standard cell libraries, which contain area, timing, power, and layout information for each logic gate.
4. Generate a netlist of gates that perform functions specified in the RTL design. The design may be optimized for area, speed, or power.

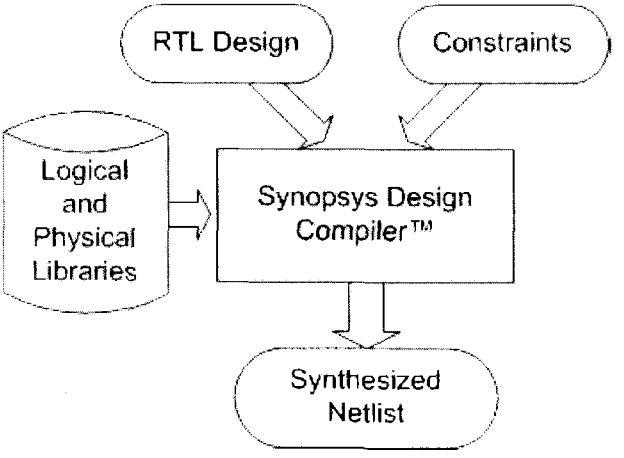

Figure 3-1. Flowchart of a typical circuit synthesis process

The following is a sample TCL code listing for synthesizing a circuit in VHDL to generate a verilog netlist.

```
# set up library
set target_library { target_library }
set link_library { link_library }

# read in VHDL
read_vhdl circuit.vhdl

# set current design
current_design design_name

# link design to library
link

# synthesize the circuit
compile

# flatten design (optional)
Ungroup -all flatten

# write netlist
write -format verilog -output netlist.v
```

3.2.2 Power Dissipation Report

The power dissipation of a design can be obtained using the *report_power* command in Design Compiler™. The *report_power* command reports the total dynamic and leakage power of a design. The dynamic power is broken down into the cell internal power, which is the short circuit power plus the charging and discharging of any internal capacitances within a gate, and the net switching power, which is the power dissipated by charging and discharging of the output load capacitances at each gate. The report further breaks down the total internal power into the internal power dissipated by all combinational circuits and the internal power dissipated by all sequential circuits. The combinational and sequential gate counts are also given in the report.

The following is a sample TCL code listing for generating a power dissipation report.

```
# set up library
set target_library { target_library }
set link_library { link_library }

# read in netlist
read_verilog netlist.v

# set current design
current_design design_name

# link design to library
link

# define clock constraint
create_clock -period clock_period_in_ns -name clock_name [get_ports
   clock_name]

# generate power report
report_power -analysis_effor high -verbose -nosplit
```

3.2.3 Cell Usage Report

Design Compiler™ can generate a report on the gate composition of a circuit using the *report_reference* command. The following is a sample TCL code listing for generating a cell reference report.

```
# set up library
set target_library { target_library }
set link_library { link_library }

# read in netlist
read_verilog netlist.v

# list all cells in the design
report_reference -nosplit
```

The report lists the names of all logic gates used to synthesize the design, the source library of each gate, the corresponding area, and the count of each gate.

3.3 Synopsys PrimeTime™

Synopsys PrimeTime™ is a tool for performing static timing analysis (STA), which is a method of computing a circuit's timing performance by checking all possible paths in the design [43]. The worst-case timing delay for the circuit determines the overall circuit performance.

An alternative timing analysis method is to perform dynamic simulation, which determines the behaviour of a circuit for a given set of input vector. Dynamic simulation checks for the logical functionality of a circuit, and the timing is sensitized by the test vector. Compared with dynamic simulation, STA is faster to perform because functional verifications are not required. In addition, the timing analysis in STA is more thorough because it checks for all timing paths in the design and not just the timing delays that are sensitized by a particular set of input vectors.

3.3.1 Timing Report

A timing report can be obtained using the *report_timing* command in PrimeTime™. The following is a sample TCL code listing for generating a timing report.

```
# set up library
set target_library { target_library }
set link_library { link_library }

# read in netlist
read_verilog netlist.v

# set current design
current_design design_name

# link design to library
link

# define clock constraint
create_clock -period clock_period_in_ns -name clock_name [get_ports
   clock_name]

# generate timing report
report_timing
```

3.3.2 Timing Path Selection

To select timing paths in a design in PrimeTime™, the *get_timing_paths* command can be used. The command selects all timing paths in the design by default, but several options are available to limit the selection of the timing paths. In particular, the *-slack_lesser_than* option is useful for selecting only the paths that violate the timing constraints.

The following is the TCL code to obtain a maximum number of 5000 timing paths that have a negative slack.

```
get_timing_path -slack_lesser_than 0 -max_paths 5000
```

3.3.3 Cell Replacement

After loading a design into PrimeTime™ and linking the design to a library, each individual cell can be swapped to another type of cell with equivalent pinouts using the

swap_cell command. This command is used to replace an HVT cell by an SVT or LVT cell during the gate partitioning optimization process.

The following is a sample TCL code to swap a cell.

```
swap_cell cell_list replacement_cell
```

After swapping in a new cell, PrimeTime™ implicitly re-links the part of the design that has been changed. The original design itself, however, remains unchanged in the memory. PrimeTime™ records changes to the netlist in a separate change list, which can be exported using the *write_changes* command, as shown below.

```
write_changes -format text -output change_list.txt
```

3.4 Modelling of Timing Delays and Power

In high-level synthesis, a circuit is modelled as a netlist of gates. Each gate has been characterized by its area, load capacitances, power, and timing information. A synthesis tool obtains the information from the standard cell libraries and predicts the timing delays and power dissipations of a circuit in a hierarchical manner, thereby reducing the analysis time. The following sections will describe the timing and power models in the Synopsys tools.

3.4.1 Timing Model

In PrimeTime™, a design is divided into a set of timing paths. The signal propagation delays along each path are calculated and checked against timing violations.

A timing path has a start point and an endpoint. A start point is a clock pin of a sequential logic or an input port of the design; an endpoint is an output port of the design where output data is captured. The total delay of a path is calculated by summing all cell and net delays in the path.

The cell delay is the timing delay from the input to the output of a gate. The standard cell libraries contain timing delay tables that list the gate delay as a function of several variables, such as the input state, the input transition time, and the output load capacitance. Since it is not possible to list the timing delay under all possible conditions, PrimeTime™ uses interpolation or extrapolation methods to estimate the timing delay of a cell when a condition is not listed in the table.

The net delay is the timing delay from the output of a cell to the input of the next cell in the timing path. The net delay is dependent upon the parasitic capacitances and resistances between cells as well as the output drive strength of the cells. During the synthesis step, layout information is usually not available. Instead of determining the actual capacitance and resistance values, PrimeTime™ estimates the net delay using statistical wire load models in the standard cell libraries.

Having determined the timing delays of each cell and each net, the total path delay can be calculated as follows:

$$\text{Path Delay} = D_{clk} + D_{clk\_net} + \Sigma D_{cell} + \Sigma D_{net}, \tag{3.1}$$

where D_{clk} is the clock source delay, D_{clk_net} is the clock network delay, D_{cell} is the cell delay, and D_{net} is the network delay.

The path delay is compared with the timing constraint to determine the path slack time. The path slack time is calculated as the difference between the path delay time and the path required time:

$$\text{Slack} = \text{Required Time} - \text{Path Delay}. \tag{3.2}$$

A positive slack indicates that the path has met the timing constraint, and a negative slack indicates a timing violation in the design.

The path required time is calculated as follows:

$$\text{Required Time} = T_{clk} + D_{clk} + D_{clk\_net} - T_{clk\_uncertainty} - T_{setup}, \tag{3.3}$$

where T_{clk} is the clock period, D_{clk} is the clock source delay, D_{clk_net} is the clock network delay, $T_{clk\_uncertainty}$ is the clock uncertainty, and T_{setup} is the register setup time.

3.4.2 Power Model

Design Compiler™ reports the internal power, switching power, and leakage power of a design. Transistor-level modelling of the design is not performed; rather, each gate is modelled as a black box with pre-determined parameters for estimating power dissipations.

The cell internal power is the power dissipated within the boundaries of a cell, which consists of the short circuit power plus the charging and discharging of any internal capacitances within a gate. The internal power is a function of the input transition time and the capacitances of a cell. An internal power lookup table for each logic gate is available in the standard cell library. Similar to timing delay calculations, Design Compiler™ uses interpolation or extrapolation methods to estimate the internal power of a cell.

The switching power of a cell is the power dissipated when charging or discharging the output load capacitances and is a function of the load capacitance and the switching activity. The output load capacitance is the sum of the input capacitance of the fanout gates plus the parasitic capacitances in the routing network, which can be estimated using the statistical wire load model. The switching activity can be determined during functional simulations. When no simulation data is available, Design Compiler™ calculates a default switching activity. The probability that the state of each primary input is at logic 1 is set to 0.5 by default, and the default toggle rate is also set to 0.5, indicating that an input toggles once per two clock cycles. The logic states and toggle rates are propagated from the primary inputs throughout the rest of the design, and the switching activity of each cell is determined. After determining the switching activity and the load capacitances, the switching power can be calculated using Eq. 3.4.

$$P_{switching} = \alpha C_{load} V_{dd}^2 f_{clk} . \quad (3.4)$$

The leakage power of a cell is dependent on the input states. For each input state, a corresponding leakage power value can be obtained from the standard cell library for each logic gate. Design Compiler™ calculates the static power of a circuit by multiplying the static power value for each state by the percentage of the total simulation time at that state. When no functional simulations are performed, the default states are determined for each gate as described above.

3.5 Summary

This chapter has described the simulation tools used in this thesis. The basic functionalities and usages have been presented as well as the sample TCL codes

corresponding to each function. The timing and power models in Synopsys tools have also been described.

4 PROPOSED TRIPLE-THRESHOLD STATIC POWER REDUCTION TECHNIQUE

This chapter presents the novel triple-threshold technique proposed in this thesis. Parts of this work have been presented in [44] and [45]. The triple-threshold standard cell libraries available from CMC have been characterized with the relative performance and leakage power. Given the cell library characterization information, a new methodology for a fine-grained triple-threshold gate partitioning technique is proposed to fully utilize the benefits of the triple-threshold technology.

4.1 Characterization of Standard Cell Libraries

The threshold voltages of the 90nm standard cell libraries are listed in Table 4-1. These threshold voltage values are the nominal values given in the 90nm design rule manual. However, since the standard cells provided by CMC are black boxes without any layout information, the actual threshold voltages may differ from the nominal values depending on the layouts of each cell.

Table 4-1. Threshold voltages of the 90nm HVT, SVT and LVT standard cell libraries

Cell Library	NMOS V_T [V]	PMOS V_T [V]
HVT	0.32	-0.36
SVT	0.24	-0.29
LVT	0.18	-0.24

To compare the performance and static power dissipation of designs synthesized with the three different standard cell libraries, a 16-bit Wallace tree multiplier with a carry look-ahead adder has been developed as a test vehicle. The 16-bit multiplier is synthesized in Synopsys Design Compiler™ to generate a netlist of 1123 cells. The design has been synthesized three times: with the HVT library only, with the SVT library only, and with the LVT library only. The static power dissipation reported by Design Compiler™ for each synthesized design is recorded, and STA is performed in PrimeTime™ for each design to determine the timing delay of the longest critical path. The results are shown in Table 4-2.

Table 4-2. Performance comparison of a 16-bit Wallace tree multiplier synthesized using the HVT, SVT and LVT standard cell libraries

Cell Library	Longest Path Delay [ns]	Max. Clock Frequency [MHz]	Static Power [μW]
HVT	3.4	294.1	0.75506
SVT	2.6	384.6	14.4600
LVT	2.1	476.2	270.7120

The 90nm standard cell libraries have been designed such that an HVT gate would occupy the same area as an SVT or an LVT gate. Therefore all three synthesized designs occupy the same die area.

The simulation results of the 16-bit multiplier indicate that the SVT synthesized design dissipates static power that is about 20 times larger than the HVT synthesized design, and the LVT design dissipates about 20 times larger static power than the SVT

design. Performance-wise, the LVT design is 24% faster than the SVT design and 62% faster than the HVT design. To fully utilize the triple-threshold technology, SVT and LVT cells must be placed in critical paths of a design to increase the clock frequency, while the number of SVT and LVT cells must be minimized due to the large penalty in the static power. The next section presents the proposed fine-grained triple-threshold gate partitioning methodology for minimizing static power while maximizing speed.

4.2 Methodology

Simulations of the 16-bit multiplier indicate that replacing an HVT cell by an SVT cell would result in a 20 times increase in the static power, and replacing an SVT cell by an LVT cell also results in a 20 times increase in the static power. Therefore only the minimum number of SVT and LVT cells should be used in a design to reduce timing delays.

The proposed methodology is an MISA-type HVT to LVT algorithm similar to the algorithm presented in [36] where it has been applied to the dual-threshold technology. As previously stated in Section 2.2.2.4, by assigning the number of critical paths passing through a gate as the weight of the gate, the algorithm may be more effective in reducing the number of critical paths with fewer cell replacements. The MISA algorithm is also less computationally intensive compared with some other algorithms and is therefore chosen as the basis for developing the proposed triple-threshold algorithm.

The TCL code used to implement the dual-threshold algorithm in [36] has been provided by Benjamin Chung at PMC Sierra. The code has been modified to work with the 90nm standard cell libraries at SFU.

To extend the dual-threshold algorithm for use with the triple-threshold technology, the usage of cells in the triple-threshold standard cell libraries must be prioritized. The priority of HVT cells is higher than SVT cells due to the lower static power dissipation, and the priority of LVT cells is the lowest because of the enormous increase in the static power. The proposed triple-threshold algorithm involves the following steps:

1. *Initial Synthesis*: The RTL design is synthesized using the HVT standard cell library to produce an HVT netlist.
2. *Incremental Replacement of SVT Cells*: STA is performed for the HVT netlist to determine the timing-critical paths in the design. A weight (or cost) is assigned to each cell in the critical paths to indicate the number of critical paths passing through the cell. The cell with the highest cost is replaced by the equivalent SVT cell. This step is repeated until the timing constraint has been met or all the HVT cells in the critical paths have been replaced with SVT cells.
3. *Incremental Replacement of LVT Cells*: If all HVT cells in the critical paths have been replaced by SVT cells and the design still violates the timing constraint, STA is performed to select critical paths, and the highest cost cells are replaced with the equivalent LVT cells. The LVT cell replacement is repeated until the timing requirement has been met.

The proposed algorithm prioritizes using HVT cells in a design first and LVT cells last, ensuring that the leakiest cells are only used if necessary while optimizing a design to meet timing requirements. Figure 4-1 shows a flowchart of the proposed triple-threshold algorithm.

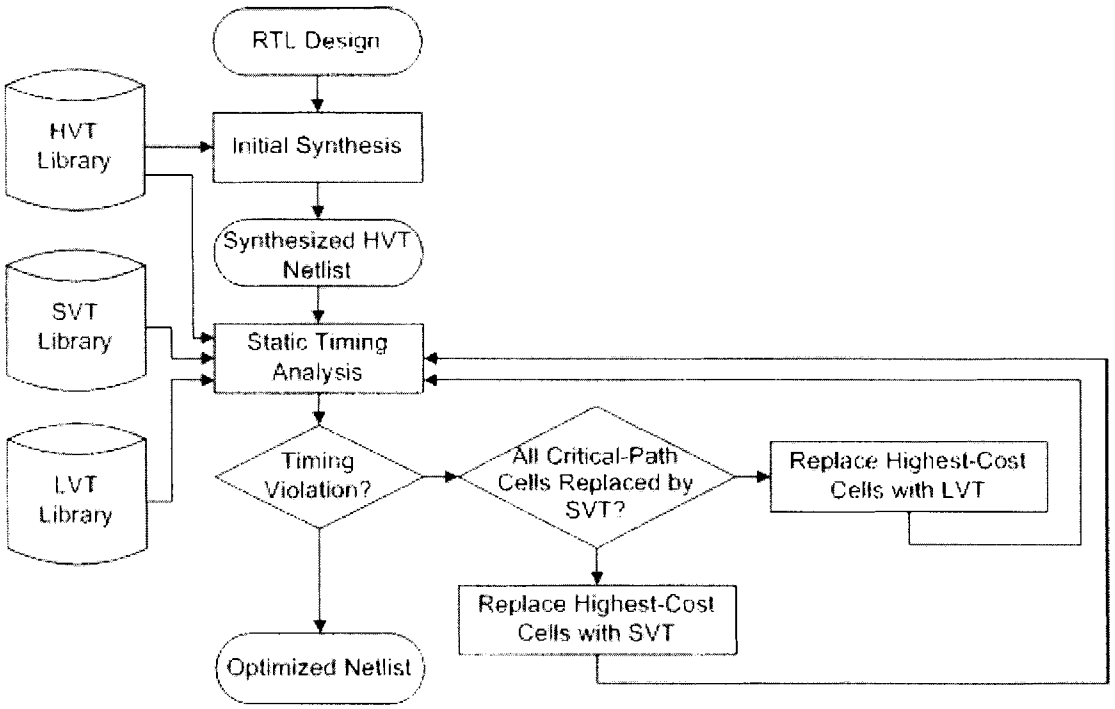

Figure 4-1. Flowchart of the proposed triple-threshold algorithm

4.2.1 Implementation Limitations

Due to the NP-hard nature of the problem, selecting all critical paths at once is impractical. For example, the total number of critical paths in the 16-bit multiplier well exceeds seven figures. On the other hand, PrimeTime™ is only capable of selecting around 500,000 paths using the *get_timing_paths* command due to memory addressing constraints on a 32-bit Sun workstation. Selecting all paths may be possible on a 64-bit workstation; however, the run time required is also exponentially large. A modification in the algorithm is necessary to address the practical limitations.

The proposed algorithm in the previous section selects the global highest-cost cells to be replaced by cells with a higher threshold voltage. Since only a limited number of paths can be selected, the local highest-cost cells within the selected paths are selected for replacement instead. Starting with an HVT netlist, the modified algorithm selects a number of critical paths from the design. If all local highest-cost cells have been replaced by SVT cells and the paths are still identified by PrimeTime™ as critical cells, one local highest-cost SVT cell is replaced by its counterpart LVT cell. The LVT cell replacement may have changed the timing delays in the design, and running STA again may result in PrimeTime™ selecting new critical paths that contain HVT cells. The algorithm must then return to the state for incremental replacement of SVT cells. Figure 4-2 shows the modified flowchart with the changes shaded in gray.

The *max_paths* variable can be set in PrimeTime™ to indicate the maximum number of timing paths to obtain during STA. This variable increases the run time significantly when set to a large number exceeding 1000 and for a large design exceeding 1000 gates. However, this variable has no direct correlation to the reductions in the static power. When *max_paths* is set to a number lower than 50, the run time may be very fast, but the resulting design may be suboptimal. Setting the *max_paths* variable to a number between 100 and 1000 results in circuits that vary slightly in the static power reduction, but the optimal number must be determined empirically for each circuit to be optimized. In general, however, the variation in the static power reduction is little and does not warrant performing extra experiments to determine the best value for *max_paths*.

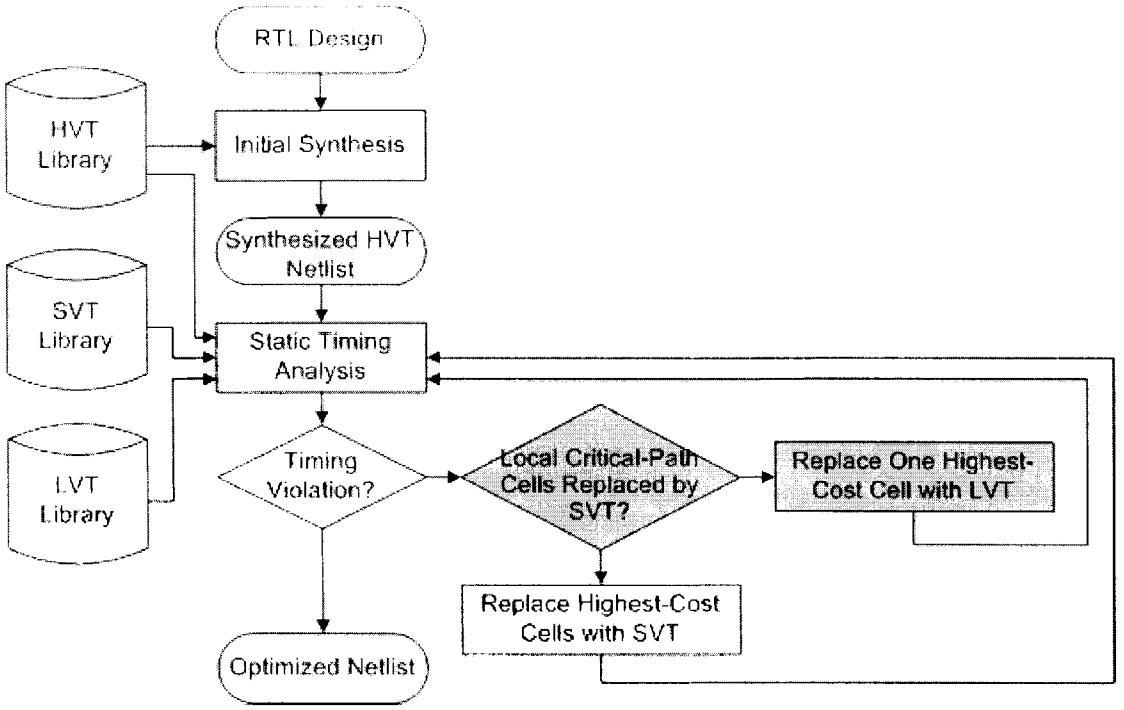

Figure 4-2. Flowchart of the modified triple-threshold algorithm

4.3 Summary

This chapter has presented the simulation results of the 16-bit multiplier as a test vehicle for characterizing the triple-threshold standard cell libraries. The results lead to the priority of using cells with a higher threshold voltage first. A suitable fine-grained triple-threshold static power minimization methodology is proposed and described. Due to practical limitations, a modified algorithm has been developed.

5 SIMULATION RESULTS

This chapter presents the experimental results of optimizing designs using the proposed triple-threshold methodology. Simulations of the 16-bit multiplier are presented in more detail. For comparison with the dual-threshold technique [36], two sets of benchmark circuit suites have been optimized. The VHDL source code of the 1995 high-level synthesis benchmark circuits is obtained from http://www.ece.vt.edu/mhsiao/hlsyn.html, and the ITC'99 benchmark suite VHDL code is obtained from http://www.ite.tul.cz/asic/iscas/index.html.

5.1 16-Bit Wallace Tree Multiplier

A 16-bit Wallace tree multiplier has been implemented as a test vehicle for experiments conducted in this thesis. The goal is to choose a common circuit that has considerable complexity and size. Since the multiplier is a fundamental circuit in processor designs, the 16-bit Wallace tree multiplier architecture is chosen for implementation.

5.1.1 Multiplier Circuit Overview

A typical multiplier circuit consists of three stages: partial product generation, partial product accumulation, and final addition [3]. The first stage generates partial products by taking the logical AND operation of the two multiplicands. For a 16-bit multiplication calculation, 16 partial products are generated, as shown in Figure 5-1. The number of partial products may be reduced using the modified Booth's recoding scheme

[46]. However, since the focus of this thesis is not on optimizing the multiplier architecture, the modified Booth's recoding is not implemented in the 16-bit multiplier. Partial products are generated using simple logical AND operations in the implementation.

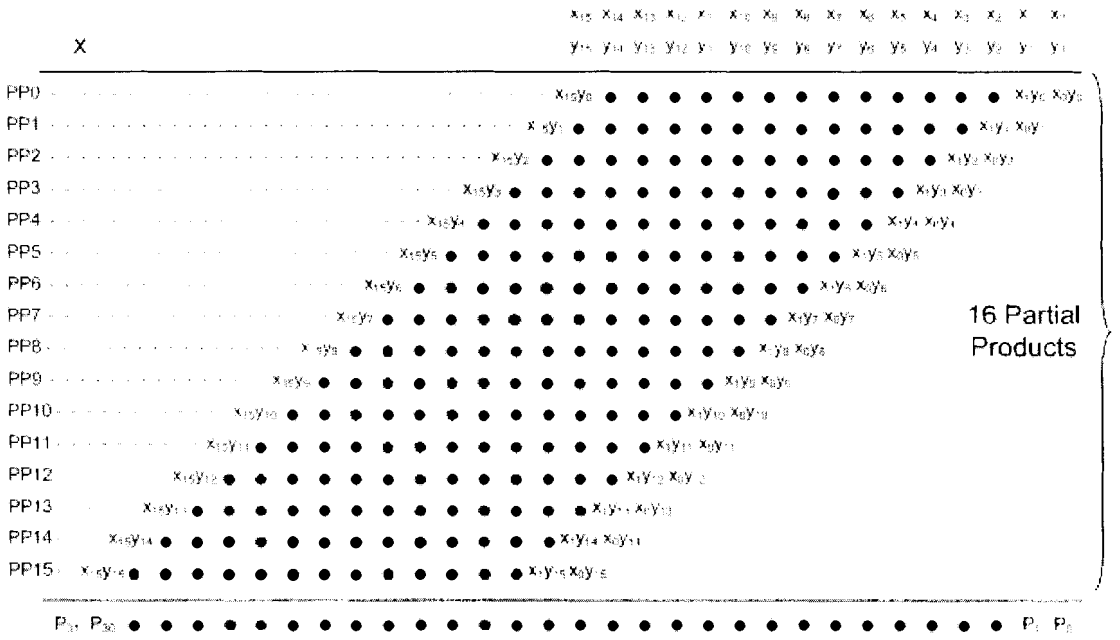

Figure 5-1. Partial products in a 16-bit multiplication

The second stage in the multiplication process is to sum the 16 partial products and simplify into two final partial products. The Wallace tree structure [47] is a simple yet fundamental architecture for implementing a fast multiplier. Using 3-2 carry-save adders, the accumulation of the partial products can be performed in a few stages without ripple delays. The block diagram of the implemented Wallace tree structure is shown in Figure 5-2.

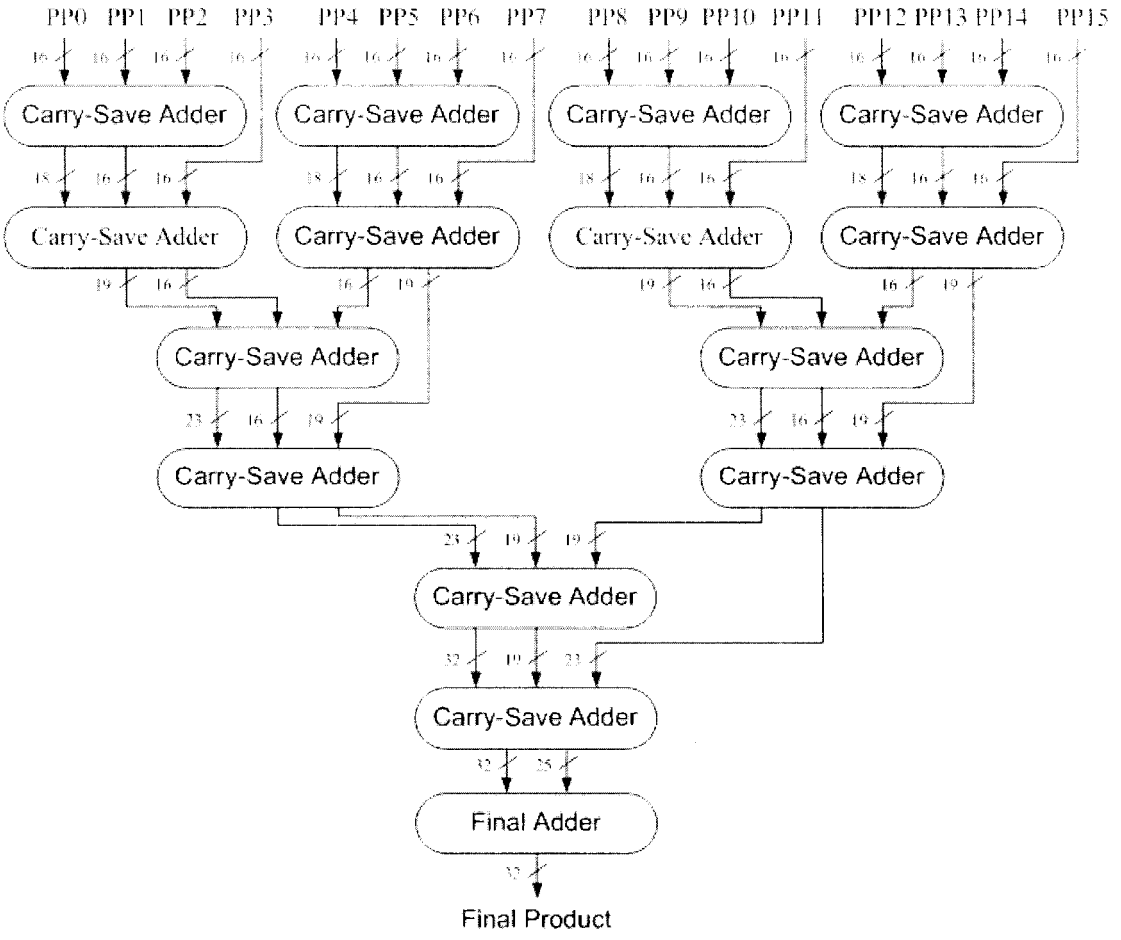

Figure 5-2. Block diagram of the Wallace tree structure

The final stage of the multiplier consists of an adder to add the two final accumulated partial products. A simple ripple adder is too slow to be considered in a multiplier architecture, while the Kogge-Stone carry-lookahead adder may add unnecessary complexity to the implementation. The monolithic carry-lookahead adder is therefore chosen for the 16-bit multiplier implementation.

The 16-bit multiplier VHDL code is listed in Appendix B.

5.1.2 Simulation Results

Table 4-2 has been duplicated below for reference. As shown in the table, the HVT multiplier has a longest path delay of 3.4 ns, whereas the LVT multiplier has a longest path delay of 2.1 ns. To generate an optimized netlist using the triple-threshold methodology, the clock constraint is set to 2.1 ns, which is the shortest path delay in the LVT design. The triple-threshold optimized netlist is compared to the dual-threshold optimized design [36]. The original netlist consists of 1123 cells; the dual-threshold optimized design consists of 897 HVT cells and 226 LVT cells; the triple-threshold optimized netlist consists of 833 HVT cells, 196 SVT cells, and 94 LVT cells.

Table 5-1. Performance comparison of a 16-bit Wallace tree multiplier synthesized using the HVT, SVT and LVT standard cell libraries

Cell Library	Longest Path Delay [ns]	Max. Clock Frequency [MHz]	Static Power [µW]
HVT	3.4	294.1	0.75506
SVT	2.6	384.6	14.4600
LVT	2.1	476.2	270.7120

Table 5-2. Static power comparison of the LVT, dual-threshold and triple-threshold designs

Multiplier Design	# HVT	# SVT	# LVT	% LVT	Static Power [µW]
LVT	0	0	1123	100%	270.71
Dual-Threshold	897	0	226	20%	59.89
Triple-Threshold	833	196	94	8.4%	27.40

By using 91.6% fewer LVT gates in the design, the triple-threshold multiplier has reduced the static power by 90% compared to the LVT multiplier. Compared to the dual-threshold multiplier, the triple-threshold multiplier contains 58.4% fewer LVT gates, and the static power dissipation is 54% less. Since LVT gates dissipate significantly more static power than HVT and SVT gates, the triple-threshold methodology has demonstrated its effectiveness in reducing the static power by minimizing the use of LVT gates in a design.

An arbitrary path from input x_7 to output P_{31} is selected to analyze the difference in static power between the dual-threshold design and the triple-threshold design. Figure 5-3 shows the path in (a) the dual-threshold optimized design and (b) the triple-threshold optimized design.

The dual-threshold path contains 9 HVT cells (shaded in gray) and 23 LVT cells (in black outline), while the triple-threshold path contains 7 HVT cells, 8 SVT cells (shaded in gray stripes), and 17 LVT cells. Comparing the two paths, nine cells have been assigned with different threshold voltages. The delay and static power for the nine cells are presented in Table 5-3.

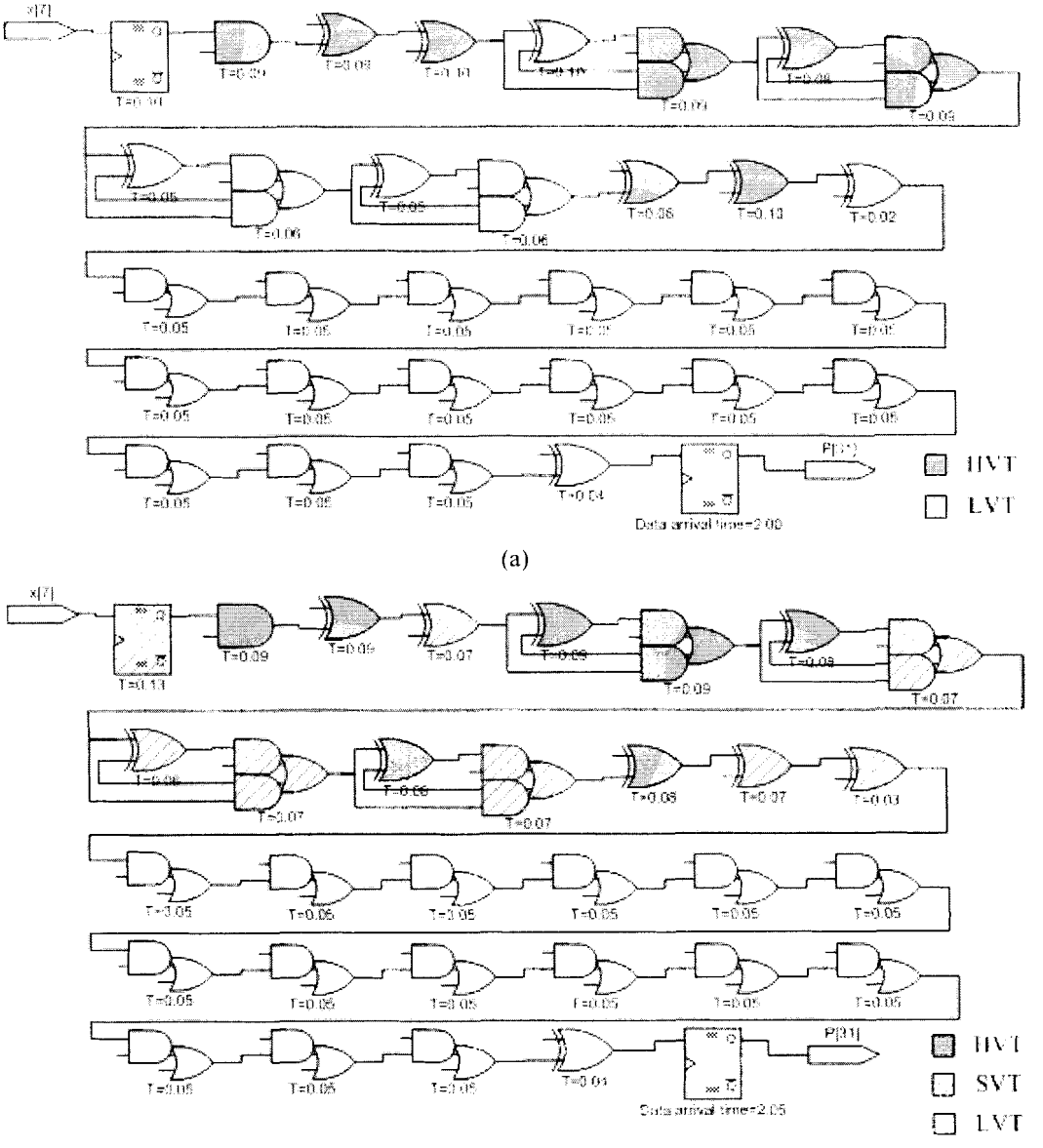

Figure 5-3. Timing path from x_7 to P_{31} in (a) dual-threshold multiplier and (b) triple-threshold multiplier

Table 5-3. Timing delay and static power of selected gates in the dual-threshold and triple-threshold paths

Dual-V_t Path Cell	Path Delay [ns]	Static Power [μW]	Tri-V_t Path Cell	Path Delay [ns]	Static Power [μW]
FD2QLVT	0.10	478.8098	FD2QSVT	0.13	24.34833
EOHVT	0.10	0.855517	EOSVT	0.08	17.25491
AO2NHVT	0.09	0.545157	AO2NSVT	0.07	10.10962
EOLVT	0.05	325.1559	EOSVT	0.06	17.5569
AO2NLVT	0.06	188.8806	AO2NSVT	0.07	10.09687
EOLVT	0.05	327.8204	EOHVT	0.08	0.898775
AO2NLVT	0.06	189.5429	AO2NSVT	0.07	10.13421
EOHVT	0.10	0.925426	EOSVT	0.08	17.96629
EOLVT	0.02	335.4581	EOSVT	0.03	18.09893
Total	0.63	1847.994	Total	0.68	126.4648

Compared to the dual-threshold path, the triple-threshold path has 6 fewer LVT cells and 8 additional SVT cells. The use of fewer LVT cells results in an increased delay of 0.05 ns. Referring to Figure 5-3, the total path delays for the dual-threshold and triple-threshold paths are 2.00 ns and 2.05 ns, respectively. Both optimized paths still meet the 2.1 ns timing constraint. Note that 0.05 ns of setup time is required; therefore the triple-threshold path has zero slack time after the optimization and the dual-threshold path has 0.05 ns of slack time. For comparison, the same path in the HVT multiplier has a total path delay of 2.79 ns. The use of LVT cells in the optimized paths has effectively reduced the circuit delay time to meet the timing constraint. Compared to the dual-threshold path, the total static power for the nine cells in the triple-threshold path is

93.2% lower. This demonstrates the effectiveness of the triple-threshold technique in reducing static power.

The triple-threshold optimized multiplier design is capable of running at the same speed as the pure LVT design, while dissipating 90% less static power. However, not all designs need to be run at the fastest possible speed. Figure 5-4 shows the static power dissipation of the same multiplier optimized with different clock constraints. At the highest clock speed (lowest clock period), the static power reduction is 90% as previously presented. As the clock speed requirement is lowered, fewer LVT gates are required in the design to meet the timing requirement, and therefore the static power dissipation is even lower. Using the triple-threshold technique, the 90% reduction in static power is the *minimum* reduction that can be achieved in the 16-bit multiplier design. At lower target clock frequencies, more than 95% static power reduction can be achieved.

The proposed triple-threshold technique not only allows designers to maximize a circuit's clock speed while minimizing static power, but also to make trade-off decisions to obtain more static power reductions by reducing speed requirements.

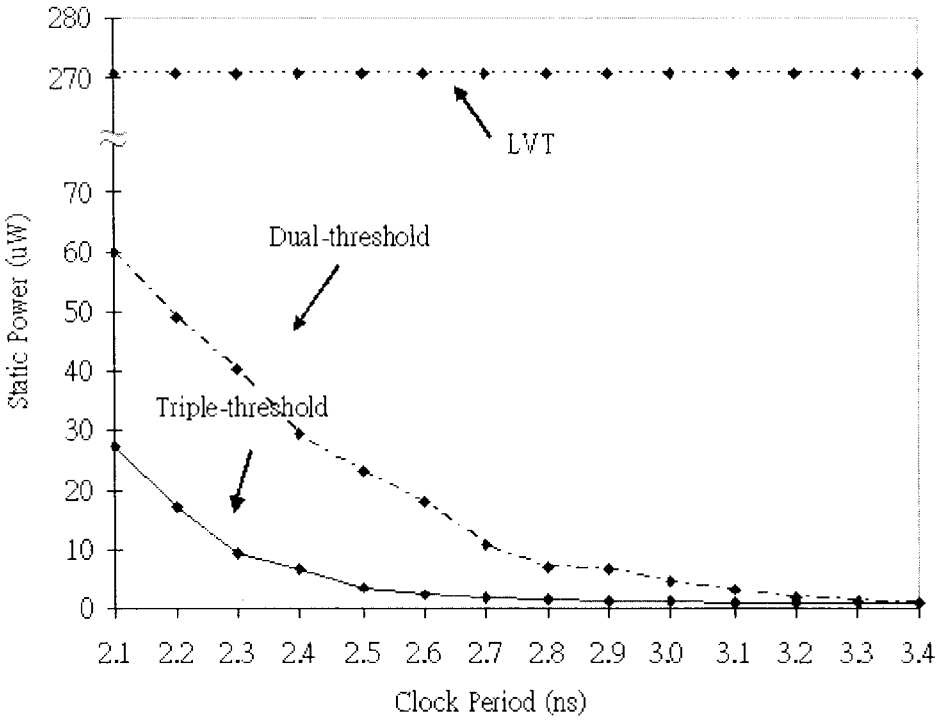

Figure 5-4. Static power dissipation of designs optimized with different clock constraints

5.2 1995 High-Level Synthesis Benchmark Circuit Suite

The 1995 high-level synthesis benchmark circuit suite consists of eight circuits. Table 5-4 summarizes the functions of each circuit. The eight circuits are optimized using the dual-threshold and triple-threshold algorithms on a Sun Ultra 45 workstation. Table 5-5 shows the static power reductions in the dual-threshold and triple-threshold optimized designs, and Table 5-6 shows the run times for optimizing the circuits.

The optimization run time for the triple-threshold algorithm is up to twice as long as the dual-threshold algorithm for small designs, but as circuit complexity increases, the triple-threshold optimization run time is comparable to the dual-threshold run time.

On average, the triple-threshold optimized designs have 89.3% reduction in static power compared to the LVT designs. Compared to the dual-threshold optimized designs,

the average reduction in static power is 41%. In all circuits tested, the triple-threshold algorithm has generated circuits with the lowest static power dissipation.

In Table 5-5, the clock period of each circuit is set to the longest critical path delay of the LVT circuit, which is the fastest clock possible for each design. As previously established, for lower clock speed requirements, more static power reductions may be achieved. Therefore the reported reduction in static power is the minimum reduction that can be achieved for each circuit.

Table 5-4. Functions of circuits in the 1995 high-level synthesis benchmark suite

Circuit	Function
am2910	Microprogram address sequencer
barcode	Barcode reader design
dhrc	Differential heat release computation circuit
diffeq	Solves a particular differential equation
gcd	Computes the greatest common divisor of two numbers
kalman	An implementation of the Kalman filter
lru	Part of a cache controller circuit that finds the least recently used item in the cache
prawn	A simple 8-bit microprocessor

Table 5-5. Static power reductions in dual-V_t and triple-V_t optimized designs

Circuit	# Gates	Clock Period [ns]	Static Power [μW]			% Saving vs. LVT		Tri-V_t % Saving vs. Dual-Vt
			LVT	2-V_t	3-V_t	2-V_t	3-V_t	
am2910	601	1.38	160.0487	14.1046	7.1088	91.2	95.6	49.6
barcode	204	0.81	57.2719	8.6383	4.6698	84.9	91.8	45.9
dhrc	1378	2.82	334.9089	42.9717	25.0963	87.2	92.5	41.6
diffeq	5465	5.00	1478.4	337.0328	181.4167	77.2	87.7	46.2
gcd	451	2.80	94.3967	27.0290	21.0709	71.4	77.7	22.0
kalman	2275	2.19	485.6676	42.3875	25.9261	91.3	94.7	38.8
lru	476	0.99	101.4512	17.5919	10.8594	82.7	89.3	38.3
prawn	733	1.12	133.3684	38.0656	19.9524	71.5	85.0	47.6
Avg.						82.2	89.3	41.3

Table 5-6. Composition of gates and optimization run time

Circuit	Dual-V_t		Tri-V_t			Run Time (h:mm:ss)	
	# HVT	# LVT	# HVT	# SVT	# LVT	Dual-V_t	Tri-V_t
am2910	541	60	513	65	23	0:00:34	0:00:59
barcode	168	36	147	40	17	0:00:13	0:00:27
dhrc	1214	164	1136	171	71	0:02:23	0:02:37
diffeq	4558	907	4218	836	411	0:52:40	1:19:17
gcd	347	104	304	68	79	0:02:17	0:04:25
kalman	2097	178	2035	139	101	0:03:50	0:07:47
lru	387	89	375	44	57	0:00:44	0:01:15
prawn	538	195	429	203	101	0:03:09	0:06:25

5.3 ITC'99 Benchmark Circuit Suite

The ITC'99 benchmark suite [51] has been developed as an update to the ISCAS'85 [49] and ISCAS'89 [50] gate-level benchmark circuits to reflect modern circuit designs. The functions of each circuit in the ITC'99 benchmark suite and simulation results of the dual-threshold and triple-threshold optimized designs are shown in Appendix A.

The smallest circuit B02 is selected to illustrate how gates of different threshold voltages are placed in the dual-threshold and triple-threshold optimized designs, as shown in Figure 5-5. The dual-threshold circuit contains 11 HVT cells and 7 LVT cells; the triple-threshold circuit contains 7 HVT cells, 7 SVT cells, and 4 LVT cells. The triple-threshold circuit dissipates 30.17% less static power compared to the dual-threshold circuit due to the use of fewer LVT cells.

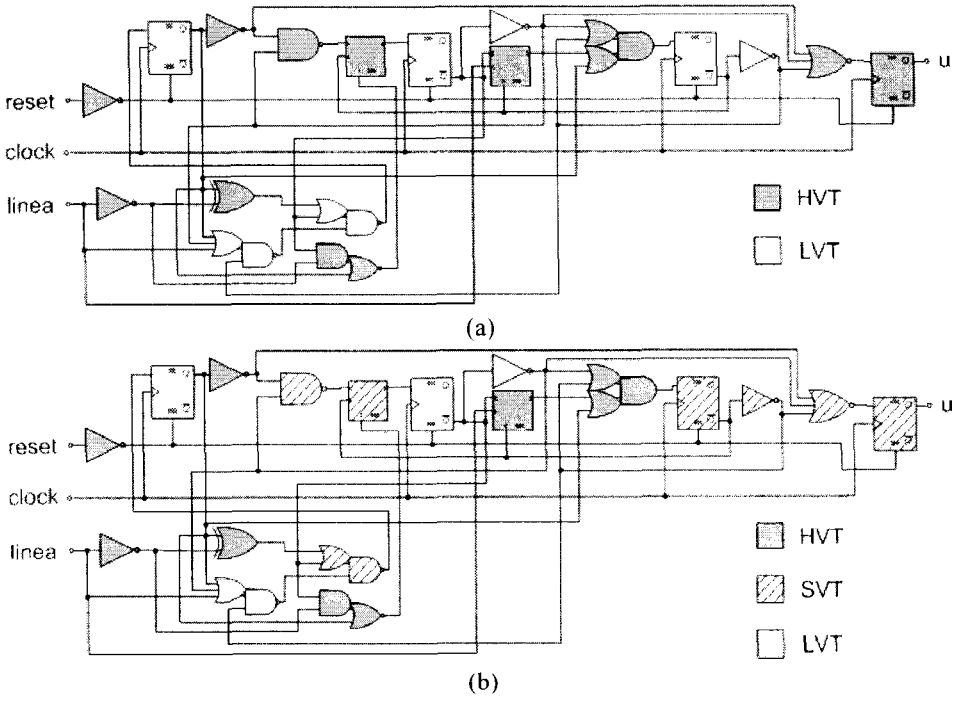

Figure 5-5. Circuit B02 optimized with (a) dual-threshold technique and (b) triple-threshold technique

Figure 5-6 and Figure 5-7 show the static power dissipations and number of LVT cells in the LVT, dual-threshold and triple-threshold designs. Comparing the two figures, it is evident that the static power dissipation follows the same trend as the number of LVT cells in a design. A direct relation between static power and the number of LVT cells can be implied.

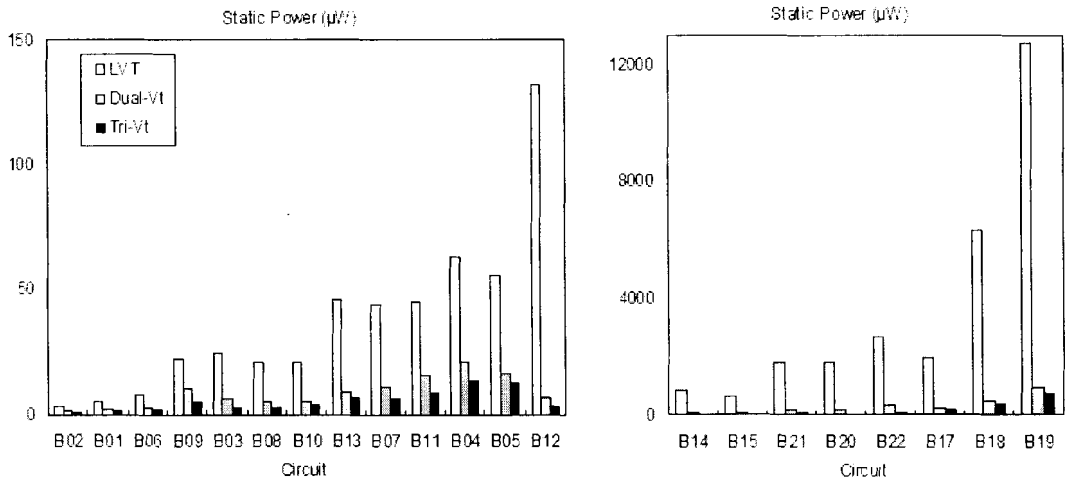

Figure 5-6. Static power of the LVT, dual-threshold, and triple-threshold designs

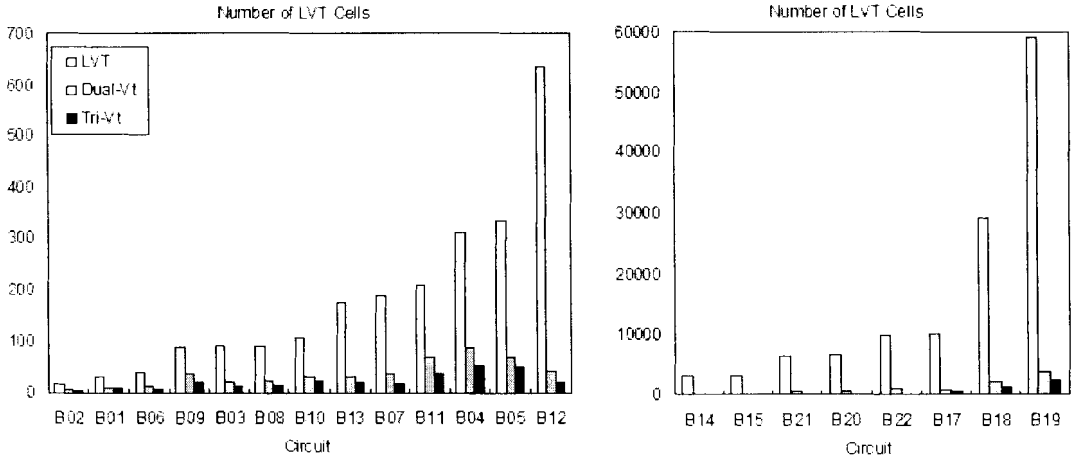

Figure 5-7. Number of LVT cells in the LVT, dual-threshold, and triple-threshold designs

Figure 5-8 plots the static power vs. the number of LVT cells. Since LVT cells are the dominant contributors to the total static power in a circuit, a linear trend can be established where static power increases directly as the number of LVT cells in a design. For each decade increase in the number of LVT cells used in a design, the static power of the design also increases by one decade, regardless of the number of HVT or SVT cells in the design. The static power can be estimated using Eq. 5.1:

$$P_{static} = \alpha \cdot N_{LVT}, \tag{5.1}$$

where α is a technology dependent parameter and equals 0.247 μW/gate for the 90nm technology provided by CMC.

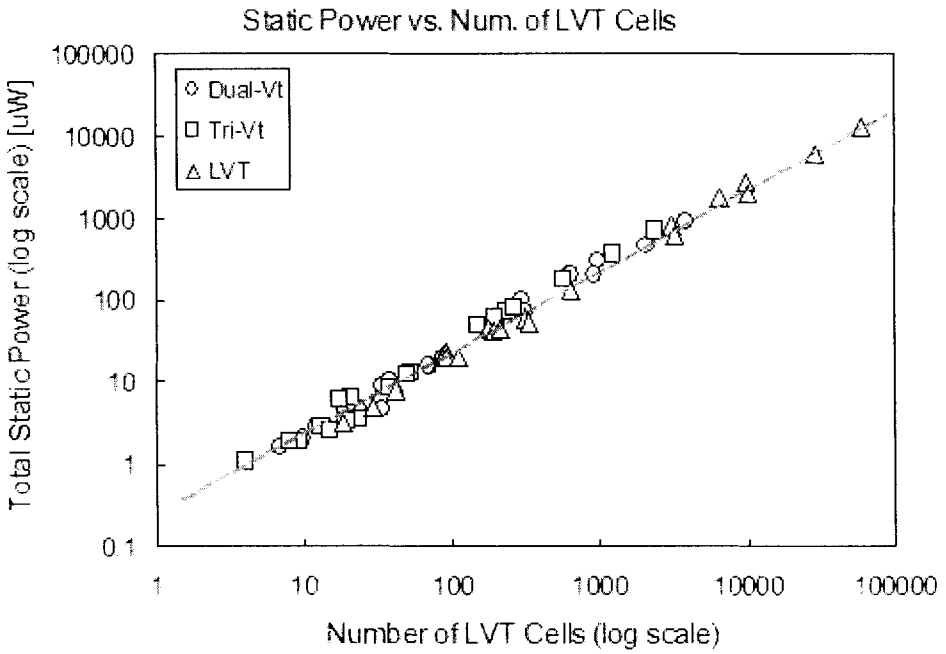

Figure 5-8. Static power vs. number of LVT cells

5.4 Summary

This chapter has presented the simulation results in this work. A 16-bit Wallace tree has been implemented as a test vehicle for synthesis with the triple-threshold standard cell libraries. A comparison has been made for the relative clock speed and static power dissipation between cells of different threshold voltages, and a 20 times difference in static power has been observed between HVT and SVT cells and between SVT and LVT cells. Based on a dual-threshold MISA gate partitioning algorithm, a suitable triple-threshold static power minimization for the 90nm technology has been proposed and implemented. A minimum static power reduction of 90% has been achieved using the proposed triple-threshold technique to optimize the 16-bit multiplier. As clock speed requirements become lower, more static power reductions can be achieved. The triple-threshold technique has also been compared to the dual-threshold technique using two sets of benchmark circuits. The optimization run time of the triple-threshold technique may be up to twice as long as the dual-threshold technique for small designs, but the difference in run time diminishes for large designs. A linear relationship has been observed between the static power dissipation in a design and the number of LVT cells being used. This observation shows that placing higher priority on using HVT cells and lower priority on SVT and LVT cells contributes to the effectiveness of the proposed triple-threshold methodology. In all benchmark circuits tested, designs optimized with the triple-threshold technique have the lowest static power dissipations.

6 CONCLUSIONS

As CMOS technology development progresses towards the nanometre regime, power management becomes a problem as more devices are integrated in a system-on-chip. With the continual scale down of power supply voltages and threshold voltages, the static power increases exponentially and becomes dominant in the total power envelope. To combat the increase in static power, extensive research has been done over the past two decades in various areas.

One of the research areas has been to utilize the MTCMOS technology and take advantage of the difference in speed and leakage current of transistors with different threshold voltages. In Chapter 2, three main topics of circuit design techniques using the MTCMOS technology have been presented, i.e. the source/body biasing, virtual supply rails, and gate partitioning techniques. The gate partitioning technique requires less design costs and provides more fine-grained control for reducing static power compared to the other techniques. Previously proposed gate partitioning techniques utilize the dual-threshold technology, and several LVT to HVT and HVT to LVT algorithms have been presented and compared. By extending the MISA-based HVT to LVT algorithm to utilize the triple-threshold CMOS technology, more fine-grained static power reduction may be possible compared to previous techniques.

Chapter 3 presents the simulation tools used in this thesis. The Synopsys tools have been used for synthesis of test circuits and generation of timing and power reports. Sample TCL scripts have also been presented.

Chapter 4 describes the proposed novel triple-threshold static power minimization methodology. The MISA-based algorithm has been adapted for use with the triple-threshold standard cell libraries. Based on the characterization of the cell libraries, a priority scheme has been set for the usage of HVT, SVT, and LVT cells. A suitable algorithm has been developed and presented. To work around limitations of the tools, a modified algorithm is used.

Benchmark circuit suites have been used to determine the effectiveness of the proposed triple-threshold algorithm. The circuits have been optimized using the proposed technique, and compared with circuits optimized with the dual-threshold technique. The simulation results have been presented in Chapter 5. For the 16-bit Wallace tree multiplier, a minimum reduction in static power of 90% has been achieved. The proposed technique allows designers to trade off clock speed with static power reduction. The optimization run times for the triple-threshold technique is comparable to the dual-threshold technique for large designs. For all circuits tested, the proposed triple-threshold technique is shown to optimize circuits with the lowest static power dissipation.

In conclusion, a novel triple-threshold static power minimization technique in high-level synthesis has been proposed. The proposed technique can be included in standard design flows with relatively low design costs, while achieving the most static power reductions compared to other gate partitioning techniques.

APPENDICES

APPENDIX A: ITC'99 BENCHMARK SUITE SIMULATION RESULTS

Table A-1. Functions of circuits in the ITC'99 benchmark suite [51]

Circuit	Function
B01	Finite state machine (FSM) comparing serial flows
B02	FSM that recognizes BCD numbers
B03	Resource arbiter
B04	Computes min and max
B05	Elaborates the contents of a memory
B06	Interrupt handler9
B07	Counts points on a straight line
B08	Find inclusions in sequences of numbers
B09	Serial to serial converter
B10	Voting system
B11	Scramble string with variable cipher
B12	One-player game for guessing a sequence
B13	Interface to meteo sensors
B14	Subset of the Viper processor
B15	Subset of the 80386 processor
B16	Parametric hard-to-initialize circuit
B17	Three copies of B15
B18	Two copies of B14 and two copies of B17
B19	Two copies of B18
B20	A copy of B14 and a modified version of B14
B21	Two copies of B14
B22	A copy of B14 and two modified versions of B14

Table A-2. Static power reductions in dual-V_t and triple-V_t optimized designs

Circuit	# Gates	Clock Period [ns]	Static Power [μW]			% Saving vs. LVT		Tri-V_t % Saving vs. Dual-Vt
			LVT	2-V_t	3-V_t	2-V_t	3-V_t	
B02	18	0.28	3.164	1.591	1.111	49.7	64.9	30.2
B01	29	0.38	4.997	2.060	1.838	58.8	63.2	10.8
B06	41	0.43	7.972	2.945	1.893	63.1	76.3	35.7
B09	87	0.65	22.071	10.427	4.754	52.8	78.5	54.4
B03	90	0.77	24.049	5.766	2.890	76.0	88.0	49.9
B08	91	0.83	20.861	4.763	2.707	77.2	87.0	43.2
B10	109	0.68	20.680	4.854	3.532	76.5	82.9	27.2
B13	175	0.53	45.712	9.004	6.457	80.3	85.9	28.3
B07	189	0.89	43.718	10.603	6.052	75.8	86.2	42.9
B11	211	1.39	45.003	15.803	8.695	64.9	80.7	45.0
B04	310	0.83	62.347	20.590	12.880	67.0	79.3	37.4
B05	334	1.36	55.328	16.042	12.325	71.0	77.7	23.2
B12	636	1.17	131.733	7.240	3.488	94.5	97.4	51.8
B14	3125	4.27	846.744	98.073	48.893	88.4	94.2	50.2
B15	3269	5.60	618.743	75.076	61.863	87.9	90.0	17.6
B21	6509	4.29	1759.100	208.238	74.567	88.2	95.8	64.2
B20	6556	4.33	1763.200	206.720	59.766	88.3	96.6	71.1
B22	9826	4.36	2679.900	317.651	82.446	88.1	96.9	74.1
B17	10081	5.67	1956.900	214.189	184.386	89.1	90.6	13.9
B18	29202	5.78	6314.900	486.794	380.093	92.3	94.0	21.9
B19	59182	5.95	12781.70	930.340	734.723	92.7	94.3	21.0
Avg.						77.3	85.7	38.8

Table A-3. Composition of gates in the optimized designs

Circuit	Total # of Gates	Dual-V_t		Tri-V_t		
		# HVT	# LVT	# HVT	# SVT	# LVT
B02	18	11	7	7	7	4
B01	29	19	10	12	8	9
B06	41	29	12	18	15	8
B09	87	49	37	54	14	19
B03	90	69	21	45	32	13
B08	91	68	23	66	10	15
B10	109	76	33	46	40	23
B13	175	142	33	119	35	21
B07	189	152	37	123	49	17
B11	211	141	70	115	58	38
B04	310	222	88	183	75	52
B05	334	265	69	251	34	49
B12	636	594	42	551	66	19
B14	3125	2826	299	2541	436	148
B15	3269	2961	308	2697	374	198
B21	6509	5883	626	5280	995	234
B20	6556	5921	635	5388	974	194
B22	9826	8856	970	8049	1513	261
B17	10081	9173	908	8362	1141	578
B18	29202	27104	2098	25660	2312	1230
B19	59182	55352	3830	53513	3237	2432

APPENDIX B: 16-BIT WALLACE TREE MULTIPLIER VHDL CODE LISTING

This section presents the code listing of the 16-bit Wallace tree multiplier. The code has been developed without realizing that the "generate" statement could be used to make the code more compact. For example, the following section of code could be replaced with a more compact generate statement.

```
u2:ubfa port map (in1(2), in2(2), in3(2), c(3), s(2));
u3:ubfa port map (in1(3), in2(3), in3(3), c(4), s(3));
u4:ubfa port map (in1(4), in2(4), in3(4), c(5), s(4));
u5:ubfa port map (in1(5), in2(5), in3(5), c(6), s(5));
u6:ubfa port map (in1(6), in2(6), in3(6), c(7), s(6));
u7:ubfa port map (in1(7), in2(7), in3(7), c(8), s(7));
u8:ubfa port map (in1(8), in2(8), in3(8), c(9), s(8));
u9:ubfa port map (in1(9), in2(9), in3(9), c(10), s(9));
```

The following code is the equivalent generate statement for the above code.

```
g1 : for i in 2 to 9 generate
   fa : ubfa port map (in1(i), in2(i), in3(i), c(i+1), s(i));
```

Using generate statements can reduce a large portion of the code size. However, changes to the input or output ports of some entities may be necessary in order to use the generate statement. Therefore, the code presented listing below has not been updated with generate statements.

```
----------------------------------
-- 16-bit Wallace tree multiplier
-- by: Harry Chen
----------------------------------

library ieee;
use  ieee.std_logic_1164.all;

entity multiplier is port (
   reset    : in  std_logic;
   clk      : in  std_logic;
   in1      : in  std_logic_vector(15 downto 0);
   in2      : in  std_logic_vector(15 downto 0);
   dout     : out std_logic_vector(31 downto 0));
end multiplier;

architecture structure of multiplier is
   component reg1 port(
      reset : in  std_logic;
      clk   : in  std_logic;
      din   : in  std_logic;
      dout  : out std_logic);
   end component;

   component reg16 port(
      reset : in  std_logic;
      clk   : in  std_logic;
      din   : in  std_logic_vector(15 downto 0);
      dout  : out std_logic_vector(15 downto 0));
   end component;

   component reg32 port(
      reset : in  std_logic;
      clk   : in  std_logic;
      din   : in  std_logic_vector(31 downto 0);
      dout  : out std_logic_vector(31 downto 0));
   end component;

   component ppg port (
      in1   : in  std_logic_vector(15 downto 0);
      in2   : in  std_logic_vector(15 downto 0);
      pp0   : out std_logic_vector(15 downto 0);
      pp1   : out std_logic_vector(16 downto 1);
      pp2   : out std_logic_vector(17 downto 2);
      pp3   : out std_logic_vector(18 downto 3);
      pp4   : out std_logic_vector(19 downto 4);
      pp5   : out std_logic_vector(20 downto 5);
      pp6   : out std_logic_vector(21 downto 6);
      pp7   : out std_logic_vector(22 downto 7);
      pp8   : out std_logic_vector(23 downto 8);
      pp9   : out std_logic_vector(24 downto 9);
      pp10  : out std_logic_vector(25 downto 10);
      pp11  : out std_logic_vector(26 downto 11);
      pp12  : out std_logic_vector(27 downto 12);
      pp13  : out std_logic_vector(28 downto 13);
      pp14  : out std_logic_vector(29 downto 14);
      pp15  : out std_logic_vector(30 downto 15));
```

```
   end component;

   component wallace port (
      pp0   : in  std_logic_vector(15 downto 0);
      pp1   : in  std_logic_vector(16 downto 1);
      pp2   : in  std_logic_vector(17 downto 2);
      pp3   : in  std_logic_vector(18 downto 3);
      pp4   : in  std_logic_vector(19 downto 4);
      pp5   : in  std_logic_vector(20 downto 5);
      pp6   : in  std_logic_vector(21 downto 6);
      pp7   : in  std_logic_vector(22 downto 7);
      pp8   : in  std_logic_vector(23 downto 8);
      pp9   : in  std_logic_vector(24 downto 9);
      pp10  : in  std_logic_vector(25 downto 10);
      pp11  : in  std_logic_vector(26 downto 11);
      pp12  : in  std_logic_vector(27 downto 12);
      pp13  : in  std_logic_vector(28 downto 13);
      pp14  : in  std_logic_vector(29 downto 14);
      pp15  : in  std_logic_vector(30 downto 15);
      wlcc  : out std_logic_vector(31 downto 7);
      wlcs  : out std_logic_vector(31 downto 0));
   end component;

   component cla port (
      x     : in  std_logic_vector ( 31 downto 0 );
      y     : in  std_logic_vector ( 31 downto 7 );
      s     : out std_logic_vector ( 31 downto 0 ));
   end component;

   signal bin1  : std_logic_vector(15 downto 0);
   signal bin2  : std_logic_vector(15 downto 0);
   signal bdout : std_logic_vector(31 downto 0);
   signal pp0   : std_logic_vector(15 downto 0);
   signal pp1   : std_logic_vector(16 downto 1);
   signal pp2   : std_logic_vector(17 downto 2);
   signal pp3   : std_logic_vector(18 downto 3);
   signal pp4   : std_logic_vector(19 downto 4);
   signal pp5   : std_logic_vector(20 downto 5);
   signal pp6   : std_logic_vector(21 downto 6);
   signal pp7   : std_logic_vector(22 downto 7);
   signal pp8   : std_logic_vector(23 downto 8);
   signal pp9   : std_logic_vector(24 downto 9);
   signal pp10  : std_logic_vector(25 downto 10);
   signal pp11  : std_logic_vector(26 downto 11);
   signal pp12  : std_logic_vector(27 downto 12);
   signal pp13  : std_logic_vector(28 downto 13);
   signal pp14  : std_logic_vector(29 downto 14);
   signal pp15  : std_logic_vector(30 downto 15);
   signal wlcs  : std_logic_vector(31 downto 0);
   signal wlcc  : std_logic_vector(31 downto 7);

begin
   uregin1: reg16 port map (reset, clk, in1, bin1);
   uregin2: reg16 port map (reset, clk, in2, bin2);

   uppg: ppg port map (
      bin1, bin2,
```

```
      pp0, pp1, pp2, pp3, pp4, pp5, pp6, pp7,
      pp8, pp9, pp10, pp11, pp12, pp13, pp14, pp15);

   uwallace: wallace port map (
      pp0, pp1, pp2, pp3, pp4, pp5, pp6, pp7,
      pp8, pp9, pp10, pp11, pp12, pp13, pp14, pp15,
      wlcc, wlcs);

   ucla: cla port map (wlcs, wlcc, bdout);

   uregdout: reg32 port map (reset, clk, bdout, dout);
end structure;


library ieee;
use ieee.std_logic_1164.all;

library work;
use work.constants.all;

entity reg16 is port (
   reset    : in  std_logic;
   clk      : in  std_logic;
   din      : in  std_logic_vector(15 downto 0);
   dout     : out std_logic_vector(15 downto 0));
end reg16;

architecture behavior of reg16 is
begin

   synch_output_data : process(reset, clk)
   begin
       if ( reset = '0' ) then
          dout <= ( others => ? ? );
       elsif ( rising_edge( clk ) ) then
          dout <= din;
       end if;
   end process;

end behavior;


library ieee;
use ieee.std_logic_1164.all;

library work;
use work.constants.all;

entity reg32 is port (
   reset    : in  std_logic;
   clk      : in  std_logic;
   din      : in  std_logic_vector(31 downto 0);
   dout     : out std_logic_vector(31 downto 0));
end reg32;

architecture behavior of reg32 is
begin
```

```
   synch_output_data : process(reset, clk)
   begin
       if ( reset = '0' ) then
          dout <= ( others => ? ? );
       elsif ( rising_edge( clk ) ) then
          dout <= din;
       end if;
   end process;

end behavior;


library ieee;
use ieee.std_logic_1164.all;

entity ppg is port (
   in1   : in  std_logic_vector(15 downto 0);
   in2   : in  std_logic_vector(15 downto 0);
   pp0   : out std_logic_vector(15 downto 0);
   pp1   : out std_logic_vector(16 downto 1);
   pp2   : out std_logic_vector(17 downto 2);
   pp3   : out std_logic_vector(18 downto 3);
   pp4   : out std_logic_vector(19 downto 4);
   pp5   : out std_logic_vector(20 downto 5);
   pp6   : out std_logic_vector(21 downto 6);
   pp7   : out std_logic_vector(22 downto 7);
   pp8   : out std_logic_vector(23 downto 8);
   pp9   : out std_logic_vector(24 downto 9);
   pp10  : out std_logic_vector(25 downto 10);
   pp11  : out std_logic_vector(26 downto 11);
   pp12  : out std_logic_vector(27 downto 12);
   pp13  : out std_logic_vector(28 downto 13);
   pp14  : out std_logic_vector(29 downto 14);
   pp15  : out std_logic_vector(30 downto 15));
end ppg;

architecture structure of ppg is
   component ppg16 port(
      in1   : in  std_logic_vector(15 downto 0);
      in2   : in  std_logic;
      ppg   : out std_logic_vector(15 downto 0));
   end component;
begin
   u0:ppg16 port map (in1, in2(0), pp0);
   u1:ppg16 port map (in1, in2(1), pp1);
   u2:ppg16 port map (in1, in2(2), pp2);
   u3:ppg16 port map (in1, in2(3), pp3);
   u4:ppg16 port map (in1, in2(4), pp4);
   u5:ppg16 port map (in1, in2(5), pp5);
   u6:ppg16 port map (in1, in2(6), pp6);
   u7:ppg16 port map (in1, in2(7), pp7);
   u8:ppg16 port map (in1, in2(8), pp8);
   u9:ppg16 port map (in1, in2(9), pp9);
   u10:ppg16 port map (in1, in2(10), pp10);
   u11:ppg16 port map (in1, in2(11), pp11);
   u12:ppg16 port map (in1, in2(12), pp12);
```

```
   u13:ppg16 port map (in1, in2(13), pp13);
   u14:ppg16 port map (in1, in2(14), pp14);
   u15:ppg16 port map (in1, in2(15), pp15);
end structure;


library ieee;
use ieee.std_logic_1164.all;

entity ppg16 is port (
   in1   : in  std_logic_vector(15 downto 0);
   in2   : in  std_logic;
   ppg   : out std_logic_vector(15 downto 0));
end ppg16;

architecture behav of ppg16 is
begin
   ppg(0) <= in1(0) and in2;
   ppg(1) <= in1(1) and in2;
   ppg(2) <= in1(2) and in2;
   ppg(3) <= in1(3) and in2;
   ppg(4) <= in1(4) and in2;
   ppg(5) <= in1(5) and in2;
   ppg(6) <= in1(6) and in2;
   ppg(7) <= in1(7) and in2;
   ppg(8) <= in1(8) and in2;
   ppg(9) <= in1(9) and in2;
   ppg(10) <= in1(10) and in2;
   ppg(11) <= in1(11) and in2;
   ppg(12) <= in1(12) and in2;
   ppg(13) <= in1(13) and in2;
   ppg(14) <= in1(14) and in2;
   ppg(15) <= in1(15) and in2;
end behav;


library ieee;
use ieee.std_logic_1164.all;

entity wallace is port (
      pp0   : in  std_logic_vector(15 downto 0);
      pp1   : in  std_logic_vector(16 downto 1);
      pp2   : in  std_logic_vector(17 downto 2);
      pp3   : in  std_logic_vector(18 downto 3);
      pp4   : in  std_logic_vector(19 downto 4);
      pp5   : in  std_logic_vector(20 downto 5);
      pp6   : in  std_logic_vector(21 downto 6);
      pp7   : in  std_logic_vector(22 downto 7);
      pp8   : in  std_logic_vector(23 downto 8);
      pp9   : in  std_logic_vector(24 downto 9);
      pp10  : in  std_logic_vector(25 downto 10);
      pp11  : in  std_logic_vector(26 downto 11);
      pp12  : in  std_logic_vector(27 downto 12);
      pp13  : in  std_logic_vector(28 downto 13);
      pp14  : in  std_logic_vector(29 downto 14);
      pp15  : in  std_logic_vector(30 downto 15);
      wlcc  : out std_logic_vector(31 downto 7);
```

```
      wlcs   : out std_logic_vector(31 downto 0));
end wallace;

architecture structure of wallace is
   component csa_15_0_16_1_17_2 port(
      in1    : in  std_logic_vector(15 downto 0);
      in2    : in  std_logic_vector(16 downto 1);
      in3    : in  std_logic_vector(17 downto 2);
      c      : out std_logic_vector(17 downto 2);
      s      : out std_logic_vector(17 downto 0));
   end component;

   component csa_19_4_20_5_21_6 port(
      in1    : in  std_logic_vector(19 downto 4);
      in2    : in  std_logic_vector(20 downto 5);
      in3    : in  std_logic_vector(21 downto 6);
      c      : out std_logic_vector(21 downto 6);
      s      : out std_logic_vector(21 downto 4));
   end component;

   component csa_23_8_24_9_25_10 port(
      in1    : in  std_logic_vector(23 downto 8);
      in2    : in  std_logic_vector(24 downto 9);
      in3    : in  std_logic_vector(25 downto 10);
      c      : out std_logic_vector(25 downto 10);
      s      : out std_logic_vector(25 downto 8));
   end component;

   component csa_27_12_28_13_29_14 port(
      in1    : in  std_logic_vector(27 downto 12);
      in2    : in  std_logic_vector(28 downto 13);
      in3    : in  std_logic_vector(29 downto 14);
      c      : out std_logic_vector(29 downto 14);
      s      : out std_logic_vector(29 downto 12));
   end component;

   component csa_17_0_17_2_18_3 port(
      in1    : in  std_logic_vector(17 downto 0);
      in2    : in  std_logic_vector(17 downto 2);
      in3    : in  std_logic_vector(18 downto 3);
      c      : out std_logic_vector(18 downto 3);
      s      : out std_logic_vector(18 downto 0));
   end component;

   component csa_21_4_21_6_22_7 port(
      in1    : in  std_logic_vector(21 downto 4);
      in2    : in  std_logic_vector(21 downto 6);
      in3    : in  std_logic_vector(22 downto 7);
      c      : out std_logic_vector(22 downto 7);
      s      : out std_logic_vector(22 downto 4));
   end component;

   component csa_25_8_25_10_26_11 port(
      in1    : in  std_logic_vector(25 downto 8);
      in2    : in  std_logic_vector(25 downto 10);
      in3    : in  std_logic_vector(26 downto 11);
      c      : out std_logic_vector(26 downto 11);
```

```
   s      : out std_logic_vector(26 downto 8));
end component;

component csa_29_12_29_14_30_15 port(
   in1    : in  std_logic_vector(29 downto 12);
   in2    : in  std_logic_vector(29 downto 14);
   in3    : in  std_logic_vector(30 downto 15);
   c      : out std_logic_vector(30 downto 15);
   s      : out std_logic_vector(30 downto 12));
end component;

component csa_18_0_18_3_22_7 port(
   in1    : in  std_logic_vector(18 downto 0);
   in2    : in  std_logic_vector(18 downto 3);
   in3    : in  std_logic_vector(22 downto 7);
   c      : out std_logic_vector(19 downto 4);
   s      : out std_logic_vector(22 downto 0));
end component;

component csa_19_4_22_0_22_4 port(
   in1    : in  std_logic_vector(19 downto 4);
   in2    : in  std_logic_vector(22 downto 0);
   in3    : in  std_logic_vector(22 downto 4);
   c      : out std_logic_vector(23 downto 5);
   s      : out std_logic_vector(22 downto 0));
end component;

component csa_26_8_26_11_30_15 port(
   in1    : in  std_logic_vector(26 downto 8);
   in2    : in  std_logic_vector(26 downto 11);
   in3    : in  std_logic_vector(30 downto 15);
   c      : out std_logic_vector(27 downto 12);
   s      : out std_logic_vector(30 downto 8));
end component;

component csa_27_12_30_8_30_12 port(
   in1    : in  std_logic_vector(27 downto 12);
   in2    : in  std_logic_vector(30 downto 8);
   in3    : in  std_logic_vector(30 downto 12);
   c      : out std_logic_vector(31 downto 13);
   s      : out std_logic_vector(30 downto 8));
end component;

component csa_22_0_23_5_31_13 port(
   in1    : in  std_logic_vector(22 downto 0);
   in2    : in  std_logic_vector(23 downto 5);
   in3    : in  std_logic_vector(31 downto 13);
   c      : out std_logic_vector(24 downto 6);
   s      : out std_logic_vector(31 downto 0));
end component;

component csa_24_6_30_8_31_0 port(
   in1    : in  std_logic_vector(24 downto 6);
   in2    : in  std_logic_vector(30 downto 8);
   in3    : in  std_logic_vector(31 downto 0);
   c      : out std_logic_vector(31 downto 7);
   s      : out std_logic_vector(31 downto 0));
```

```
   end component;

   signal c0 : std_logic_vector ( 17 downto 2 );
   signal s0 : std_logic_vector ( 17 downto 0 );
   signal c1 : std_logic_vector ( 21 downto 6 );
   signal s1 : std_logic_vector ( 21 downto 4 );
   signal c2 : std_logic_vector ( 25 downto 10 );
   signal s2 : std_logic_vector ( 25 downto 8 );
   signal c3 : std_logic_vector ( 29 downto 14 );
   signal s3 : std_logic_vector ( 29 downto 12 );
   signal c4 : std_logic_vector ( 18 downto 3 );
   signal s4 : std_logic_vector ( 18 downto 0 );
   signal c5 : std_logic_vector ( 22 downto 7 );
   signal s5 : std_logic_vector ( 22 downto 4 );
   signal c6 : std_logic_vector ( 26 downto 11 );
   signal s6 : std_logic_vector ( 26 downto 8 );
   signal c7 : std_logic_vector ( 30 downto 15 );
   signal s7 : std_logic_vector ( 30 downto 12 );
   signal c8 : std_logic_vector ( 19 downto 4 );
   signal s8 : std_logic_vector ( 22 downto 0 );
   signal c9 : std_logic_vector ( 23 downto 5 );
   signal s9 : std_logic_vector ( 22 downto 0 );
   signal c10 : std_logic_vector ( 27 downto 12 );
   signal s10 : std_logic_vector ( 30 downto 8 );
   signal c11 : std_logic_vector ( 31 downto 13 );
   signal s11 : std_logic_vector ( 30 downto 8 );
   signal c12 : std_logic_vector ( 24 downto 6 );
   signal s12 : std_logic_vector ( 31 downto 0 );
begin
   u0:csa_15_0_16_1_17_2 port map ( pp0, pp1, pp2, c0, s0 );
   u1:csa_19_4_20_5_21_6 port map ( pp4, pp5, pp6, c1, s1 );
   u2:csa_23_8_24_9_25_10 port map ( pp8, pp9, pp10, c2, s2 );
   u3:csa_27_12_28_13_29_14 port map ( pp12, pp13, pp14, c3, s3 );

   u4:csa_17_0_17_2_18_3 port map ( s0, c0, pp3, c4, s4 );
   u5:csa_21_4_21_6_22_7 port map ( s1, c1, pp7, c5, s5 );
   u6:csa_25_8_25_10_26_11 port map ( s2, c2, pp11, c6, s6 );
   u7:csa_29_12_29_14_30_15 port map ( s3, c3, pp15, c7, s7 );

   u8:csa_18_0_18_3_22_7 port map ( s4, c4, c5, c8, s8 );
   u9:csa_19_4_22_0_22_4 port map ( c8, s8, s5, c9, s9 );
   u10:csa_26_8_26_11_30_15 port map ( s6, c6, c7, c10, s10 );
   u11:csa_27_12_30_8_30_12 port map ( c10, s10, s7, c11, s11 );

   u12:csa_22_0_23_5_31_13 port map ( s9, c9, c11, c12, s12 );
   u13:csa_24_6_30_8_31_0 port map ( c12, s11, s12, wlcc, wlcs );
end structure;


library ieee;
use ieee.std_logic_1164.all;

entity csa_15_0_16_1_17_2 is port (
   in1    : in  std_logic_vector(15 downto 0);
   in2    : in  std_logic_vector(16 downto 1);
   in3    : in  std_logic_vector(17 downto 2);
   c      : out std_logic_vector(17 downto 2);
```

```
   s     : out std_logic_vector(17 downto 0));
end csa_15_0_16_1_17_2;

architecture structure of csa_15_0_16_1_17_2 is
   component ubha port(
      x : in  std_logic;
      y : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

   component ubfa port(
      x : in  std_logic;
      y : in  std_logic;
      z : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

begin
   s(0) <= in1(0);
   u1:ubha port map (in1(1), in2(1), c(2), s(1));
   u2:ubfa port map (in1(2), in2(2), in3(2), c(3), s(2));
   u3:ubfa port map (in1(3), in2(3), in3(3), c(4), s(3));
   u4:ubfa port map (in1(4), in2(4), in3(4), c(5), s(4));
   u5:ubfa port map (in1(5), in2(5), in3(5), c(6), s(5));
   u6:ubfa port map (in1(6), in2(6), in3(6), c(7), s(6));
   u7:ubfa port map (in1(7), in2(7), in3(7), c(8), s(7));
   u8:ubfa port map (in1(8), in2(8), in3(8), c(9), s(8));
   u9:ubfa port map (in1(9), in2(9), in3(9), c(10), s(9));
   u10:ubfa port map (in1(10), in2(10), in3(10), c(11), s(10));
   u11:ubfa port map (in1(11), in2(11), in3(11), c(12), s(11));
   u12:ubfa port map (in1(12), in2(12), in3(12), c(13), s(12));
   u13:ubfa port map (in1(13), in2(13), in3(13), c(14), s(13));
   u14:ubfa port map (in1(14), in2(14), in3(14), c(15), s(14));
   u15:ubfa port map (in1(15), in2(15), in3(15), c(16), s(15));
   u16:ubha port map (in2(16), in3(16), c(17), s(16));
   s(17) <= in3(17);
end structure;


library ieee;
use ieee.std_logic_1164.all;

entity csa_17_0_17_2_18_3 is port (
   in1   : in  std_logic_vector(17 downto 0);
   in2   : in  std_logic_vector(17 downto 2);
   in3   : in  std_logic_vector(18 downto 3);
   c     : out std_logic_vector(18 downto 3);
   s     : out std_logic_vector(18 downto 0));
end csa_17_0_17_2_18_3;

architecture structure of csa_17_0_17_2_18_3 is
   component ubha port(
      x : in  std_logic;
      y : in  std_logic;
      c : out std_logic;
```

```
      s : out std_logic);
   end component;

   component ubfa port(
      x : in  std_logic;
      y : in  std_logic;
      z : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

begin
   s(0) <= in1(0);
   s(1) <= in1(1);
   u2:ubha port map (in1(2), in2(2), c(3), s(2));
   u3:ubfa port map (in1(3), in2(3), in3(3), c(4), s(3));
   u4:ubfa port map (in1(4), in2(4), in3(4), c(5), s(4));
   u5:ubfa port map (in1(5), in2(5), in3(5), c(6), s(5));
   u6:ubfa port map (in1(6), in2(6), in3(6), c(7), s(6));
   u7:ubfa port map (in1(7), in2(7), in3(7), c(8), s(7));
   u8:ubfa port map (in1(8), in2(8), in3(8), c(9), s(8));
   u9:ubfa port map (in1(9), in2(9), in3(9), c(10), s(9));
   u10:ubfa port map (in1(10), in2(10), in3(10), c(11), s(10));
   u11:ubfa port map (in1(11), in2(11), in3(11), c(12), s(11));
   u12:ubfa port map (in1(12), in2(12), in3(12), c(13), s(12));
   u13:ubfa port map (in1(13), in2(13), in3(13), c(14), s(13));
   u14:ubfa port map (in1(14), in2(14), in3(14), c(15), s(14));
   u15:ubfa port map (in1(15), in2(15), in3(15), c(16), s(15));
   u16:ubfa port map (in1(16), in2(16), in3(16), c(17), s(16));
   u17:ubfa port map (in1(17), in2(17), in3(17), c(18), s(17));
   s(18) <= in3(18);
end structure;


library ieee;
use ieee.std_logic_1164.all;

entity csa_18_0_18_3_22_7 is port (
   in1   : in  std_logic_vector(18 downto 0);
   in2   : in  std_logic_vector(18 downto 3);
   in3   : in  std_logic_vector(22 downto 7);
   c     : out std_logic_vector(19 downto 4);
   s     : out std_logic_vector(22 downto 0));
end csa_18_0_18_3_22_7;

architecture structure of csa_18_0_18_3_22_7 is
   component ubha port(
      x : in  std_logic;
      y : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

   component ubfa port(
      x : in  std_logic;
      y : in  std_logic;
      z : in  std_logic;
```

```
      c : out std_logic;
      s : out std_logic);
   end component;

begin
   s(0) <= in1(0);
   s(1) <= in1(1);
   s(2) <= in1(2);
   u3:ubha port map (in1(3), in2(3), c(4), s(3));
   u4:ubha port map (in1(4), in2(4), c(5), s(4));
   u5:ubha port map (in1(5), in2(5), c(6), s(5));
   u6:ubha port map (in1(6), in2(6), c(7), s(6));
   u7:ubfa port map (in1(7), in2(7), in3(7), c(8), s(7));
   u8:ubfa port map (in1(8), in2(8), in3(8), c(9), s(8));
   u9:ubfa port map (in1(9), in2(9), in3(9), c(10), s(9));
   u10:ubfa port map (in1(10), in2(10), in3(10), c(11), s(10));
   u11:ubfa port map (in1(11), in2(11), in3(11), c(12), s(11));
   u12:ubfa port map (in1(12), in2(12), in3(12), c(13), s(12));
   u13:ubfa port map (in1(13), in2(13), in3(13), c(14), s(13));
   u14:ubfa port map (in1(14), in2(14), in3(14), c(15), s(14));
   u15:ubfa port map (in1(15), in2(15), in3(15), c(16), s(15));
   u16:ubfa port map (in1(16), in2(16), in3(16), c(17), s(16));
   u17:ubfa port map (in1(17), in2(17), in3(17), c(18), s(17));
   u18:ubfa port map (in1(18), in2(18), in3(18), c(19), s(18));
   s(19) <= in3(19);
   s(20) <= in3(20);
   s(21) <= in3(21);
   s(22) <= in3(22);
end structure;


library ieee;
use ieee.std_logic_1164.all;

entity csa_19_4_20_5_21_6 is port (
   in1   : in  std_logic_vector(19 downto 4);
   in2   : in  std_logic_vector(20 downto 5);
   in3   : in  std_logic_vector(21 downto 6);
   c     : out std_logic_vector(21 downto 6);
   s     : out std_logic_vector(21 downto 4));
end csa_19_4_20_5_21_6;

architecture structure of csa_19_4_20_5_21_6 is
   component ubha port(
      x : in  std_logic;
      y : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

   component ubfa port(
      x : in  std_logic;
      y : in  std_logic;
      z : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;
```

```
begin
   s(4) <= in1(4);
   u5:ubha port map (in1(5), in2(5), c(6), s(5));
   u6:ubfa port map (in1(6), in2(6), in3(6), c(7), s(6));
   u7:ubfa port map (in1(7), in2(7), in3(7), c(8), s(7));
   u8:ubfa port map (in1(8), in2(8), in3(8), c(9), s(8));
   u9:ubfa port map (in1(9), in2(9), in3(9), c(10), s(9));
   u10:ubfa port map (in1(10), in2(10), in3(10), c(11), s(10));
   u11:ubfa port map (in1(11), in2(11), in3(11), c(12), s(11));
   u12:ubfa port map (in1(12), in2(12), in3(12), c(13), s(12));
   u13:ubfa port map (in1(13), in2(13), in3(13), c(14), s(13));
   u14:ubfa port map (in1(14), in2(14), in3(14), c(15), s(14));
   u15:ubfa port map (in1(15), in2(15), in3(15), c(16), s(15));
   u16:ubfa port map (in1(16), in2(16), in3(16), c(17), s(16));
   u17:ubfa port map (in1(17), in2(17), in3(17), c(18), s(17));
   u18:ubfa port map (in1(18), in2(18), in3(18), c(19), s(18));
   u19:ubfa port map (in1(19), in2(19), in3(19), c(20), s(19));
   u20:ubha port map (in2(20), in3(20), c(21), s(20));
   s(21) <= in3(21);
end structure;


library ieee;
use ieee.std_logic_1164.all;

entity csa_19_4_22_0_22_4 is port (
   in1   : in  std_logic_vector(19 downto 4);
   in2   : in  std_logic_vector(22 downto 0);
   in3   : in  std_logic_vector(22 downto 4);
   c     : out std_logic_vector(23 downto 5);
   s     : out std_logic_vector(22 downto 0));
end csa_19_4_22_0_22_4;

architecture structure of csa_19_4_22_0_22_4 is
   component ubha port(
      x : in  std_logic;
      y : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

   component ubfa port(
      x : in  std_logic;
      y : in  std_logic;
      z : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

begin
   s(0) <= in2(0);
   s(1) <= in2(1);
   s(2) <= in2(2);
   s(3) <= in2(3);
   u4:ubfa port map (in1(4), in2(4), in3(4), c(5), s(4));
   u5:ubfa port map (in1(5), in2(5), in3(5), c(6), s(5));
```

```
   u6:ubfa port map (in1(6), in2(6), in3(6), c(7), s(6));
   u7:ubfa port map (in1(7), in2(7), in3(7), c(8), s(7));
   u8:ubfa port map (in1(8), in2(8), in3(8), c(9), s(8));
   u9:ubfa port map (in1(9), in2(9), in3(9), c(10), s(9));
   u10:ubfa port map (in1(10), in2(10), in3(10), c(11), s(10));
   u11:ubfa port map (in1(11), in2(11), in3(11), c(12), s(11));
   u12:ubfa port map (in1(12), in2(12), in3(12), c(13), s(12));
   u13:ubfa port map (in1(13), in2(13), in3(13), c(14), s(13));
   u14:ubfa port map (in1(14), in2(14), in3(14), c(15), s(14));
   u15:ubfa port map (in1(15), in2(15), in3(15), c(16), s(15));
   u16:ubfa port map (in1(16), in2(16), in3(16), c(17), s(16));
   u17:ubfa port map (in1(17), in2(17), in3(17), c(18), s(17));
   u18:ubfa port map (in1(18), in2(18), in3(18), c(19), s(18));
   u19:ubfa port map (in1(19), in2(19), in3(19), c(20), s(19));
   u20:ubha port map (in2(20), in3(20), c(21), s(20));
   u21:ubha port map (in2(21), in3(21), c(22), s(21));
   u22:ubha port map (in2(22), in3(22), c(23), s(22));
end structure;


library ieee;
use ieee.std_logic_1164.all;

entity csa_21_4_21_6_22_7 is port (
   in1   : in  std_logic_vector(21 downto 4);
   in2   : in  std_logic_vector(21 downto 6);
   in3   : in  std_logic_vector(22 downto 7);
   c     : out std_logic_vector(22 downto 7);
   s     : out std_logic_vector(22 downto 4));
end csa_21_4_21_6_22_7;

architecture structure of csa_21_4_21_6_22_7 is
   component ubha port(
      x : in  std_logic;
      y : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

   component ubfa port(
      x : in  std_logic;
      y : in  std_logic;
      z : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

begin
   s(4) <= in1(4);
   s(5) <= in1(5);
   u6:ubha port map (in1(6), in2(6), c(7), s(6));
   u7:ubfa port map (in1(7), in2(7), in3(7), c(8), s(7));
   u8:ubfa port map (in1(8), in2(8), in3(8), c(9), s(8));
   u9:ubfa port map (in1(9), in2(9), in3(9), c(10), s(9));
   u10:ubfa port map (in1(10), in2(10), in3(10), c(11), s(10));
   u11:ubfa port map (in1(11), in2(11), in3(11), c(12), s(11));
   u12:ubfa port map (in1(12), in2(12), in3(12), c(13), s(12));
```

```
   u13:ubfa port map (in1(13), in2(13), in3(13), c(14), s(13));
   u14:ubfa port map (in1(14), in2(14), in3(14), c(15), s(14));
   u15:ubfa port map (in1(15), in2(15), in3(15), c(16), s(15));
   u16:ubfa port map (in1(16), in2(16), in3(16), c(17), s(16));
   u17:ubfa port map (in1(17), in2(17), in3(17), c(18), s(17));
   u18:ubfa port map (in1(18), in2(18), in3(18), c(19), s(18));
   u19:ubfa port map (in1(19), in2(19), in3(19), c(20), s(19));
   u20:ubfa port map (in1(20), in2(20), in3(20), c(21), s(20));
   u21:ubfa port map (in1(21), in2(21), in3(21), c(22), s(21));
   s(22) <= in3(22);
end structure;


library ieee;
use ieee.std_logic_1164.all;

entity csa_22_0_23_5_31_13 is port (
   in1   : in  std_logic_vector(22 downto 0);
   in2   : in  std_logic_vector(23 downto 5);
   in3   : in  std_logic_vector(31 downto 13);
   c     : out std_logic_vector(24 downto 6);
   s     : out std_logic_vector(31 downto 0));
end csa_22_0_23_5_31_13;

architecture structure of csa_22_0_23_5_31_13 is
   component ubha port(
      x : in  std_logic;
      y : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

   component ubfa port(
      x : in  std_logic;
      y : in  std_logic;
      z : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

begin
   s(0) <= in1(0);
   s(1) <= in1(1);
   s(2) <= in1(2);
   s(3) <= in1(3);
   s(4) <= in1(4);
   u5:ubha port map (in1(5), in2(5), c(6), s(5));
   u6:ubha port map (in1(6), in2(6), c(7), s(6));
   u7:ubha port map (in1(7), in2(7), c(8), s(7));
   u8:ubha port map (in1(8), in2(8), c(9), s(8));
   u9:ubha port map (in1(9), in2(9), c(10), s(9));
   u10:ubha port map (in1(10), in2(10), c(11), s(10));
   u11:ubha port map (in1(11), in2(11), c(12), s(11));
   u12:ubha port map (in1(12), in2(12), c(13), s(12));
   u13:ubfa port map (in1(13), in2(13), in3(13), c(14), s(13));
   u14:ubfa port map (in1(14), in2(14), in3(14), c(15), s(14));
   u15:ubfa port map (in1(15), in2(15), in3(15), c(16), s(15));
```

```
   u16:ubfa port map (in1(16), in2(16), in3(16), c(17), s(16));
   u17:ubfa port map (in1(17), in2(17), in3(17), c(18), s(17));
   u18:ubfa port map (in1(18), in2(18), in3(18), c(19), s(18));
   u19:ubfa port map (in1(19), in2(19), in3(19), c(20), s(19));
   u20:ubfa port map (in1(20), in2(20), in3(20), c(21), s(20));
   u21:ubfa port map (in1(21), in2(21), in3(21), c(22), s(21));
   u22:ubfa port map (in1(22), in2(22), in3(22), c(23), s(22));
   u23:ubha port map (in2(23), in3(23), c(24), s(23));
   s(24) <= in3(24);
   s(25) <= in3(25);
   s(26) <= in3(26);
   s(27) <= in3(27);
   s(28) <= in3(28);
   s(29) <= in3(29);
   s(30) <= in3(30);
   s(31) <= in3(31);
end structure;


library ieee;
use ieee.std_logic_1164.all;

entity csa_23_8_24_9_25_10 is port (
   in1   : in  std_logic_vector(23 downto 8);
   in2   : in  std_logic_vector(24 downto 9);
   in3   : in  std_logic_vector(25 downto 10);
   c     : out std_logic_vector(25 downto 10);
   s     : out std_logic_vector(25 downto 8));
end csa_23_8_24_9_25_10;

architecture structure of csa_23_8_24_9_25_10 is
   component ubha port(
      x : in  std_logic;
      y : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

   component ubfa port(
      x : in  std_logic;
      y : in  std_logic;
      z : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

begin
   s(8) <= in1(8);
   u9:ubha port map (in1(9), in2(9), c(10), s(9));
   u10:ubfa port map (in1(10), in2(10), in3(10), c(11), s(10));
   u11:ubfa port map (in1(11), in2(11), in3(11), c(12), s(11));
   u12:ubfa port map (in1(12), in2(12), in3(12), c(13), s(12));
   u13:ubfa port map (in1(13), in2(13), in3(13), c(14), s(13));
   u14:ubfa port map (in1(14), in2(14), in3(14), c(15), s(14));
   u15:ubfa port map (in1(15), in2(15), in3(15), c(16), s(15));
   u16:ubfa port map (in1(16), in2(16), in3(16), c(17), s(16));
   u17:ubfa port map (in1(17), in2(17), in3(17), c(18), s(17));
```

```
   u18:ubfa port map (in1(18), in2(18), in3(18), c(19), s(18));
   u19:ubfa port map (in1(19), in2(19), in3(19), c(20), s(19));
   u20:ubfa port map (in1(20), in2(20), in3(20), c(21), s(20));
   u21:ubfa port map (in1(21), in2(21), in3(21), c(22), s(21));
   u22:ubfa port map (in1(22), in2(22), in3(22), c(23), s(22));
   u23:ubfa port map (in1(23), in2(23), in3(23), c(24), s(23));
   u24:ubha port map (in2(24), in3(24), c(25), s(24));
   s(25) <= in3(25);
end structure;


library ieee;
use ieee.std_logic_1164.all;

entity csa_24_6_30_8_31_0 is port (
   in1   : in  std_logic_vector(24 downto 6);
   in2   : in  std_logic_vector(30 downto 8);
   in3   : in  std_logic_vector(31 downto 0);
   c     : out std_logic_vector(31 downto 7);
   s     : out std_logic_vector(31 downto 0));
end csa_24_6_30_8_31_0;

architecture structure of csa_24_6_30_8_31_0 is
   component ubha port(
      x : in  std_logic;
      y : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

   component ubfa port(
      x : in  std_logic;
      y : in  std_logic;
      z : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

begin
   s(0) <= in3(0);
   s(1) <= in3(1);
   s(2) <= in3(2);
   s(3) <= in3(3);
   s(4) <= in3(4);
   s(5) <= in3(5);
   u6:ubha port map (in1(6), in3(6), c(7), s(6));
   u7:ubha port map (in1(7), in3(7), c(8), s(7));
   u8:ubfa port map (in1(8), in2(8), in3(8), c(9), s(8));
   u9:ubfa port map (in1(9), in2(9), in3(9), c(10), s(9));
   u10:ubfa port map (in1(10), in2(10), in3(10), c(11), s(10));
   u11:ubfa port map (in1(11), in2(11), in3(11), c(12), s(11));
   u12:ubfa port map (in1(12), in2(12), in3(12), c(13), s(12));
   u13:ubfa port map (in1(13), in2(13), in3(13), c(14), s(13));
   u14:ubfa port map (in1(14), in2(14), in3(14), c(15), s(14));
   u15:ubfa port map (in1(15), in2(15), in3(15), c(16), s(15));
   u16:ubfa port map (in1(16), in2(16), in3(16), c(17), s(16));
   u17:ubfa port map (in1(17), in2(17), in3(17), c(18), s(17));
```

```
   u18:ubfa port map (in1(18), in2(18), in3(18), c(19), s(18));
   u19:ubfa port map (in1(19), in2(19), in3(19), c(20), s(19));
   u20:ubfa port map (in1(20), in2(20), in3(20), c(21), s(20));
   u21:ubfa port map (in1(21), in2(21), in3(21), c(22), s(21));
   u22:ubfa port map (in1(22), in2(22), in3(22), c(23), s(22));
   u23:ubfa port map (in1(23), in2(23), in3(23), c(24), s(23));
   u24:ubfa port map (in1(24), in2(24), in3(24), c(25), s(24));
   u25:ubha port map (in2(25), in3(25), c(26), s(25));
   u26:ubha port map (in2(26), in3(26), c(27), s(26));
   u27:ubha port map (in2(27), in3(27), c(28), s(27));
   u28:ubha port map (in2(28), in3(28), c(29), s(28));
   u29:ubha port map (in2(29), in3(29), c(30), s(29));
   u30:ubha port map (in2(30), in3(30), c(31), s(30));
   s(31) <= in3(31);
end structure;


library ieee;
use ieee.std_logic_1164.all;

entity csa_25_8_25_10_26_11 is port (
   in1   : in  std_logic_vector(25 downto 8);
   in2   : in  std_logic_vector(25 downto 10);
   in3   : in  std_logic_vector(26 downto 11);
   c     : out std_logic_vector(26 downto 11);
   s     : out std_logic_vector(26 downto 8));
end csa_25_8_25_10_26_11;

architecture structure of csa_25_8_25_10_26_11 is
   component ubha port(
      x : in  std_logic;
      y : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

   component ubfa port(
      x : in  std_logic;
      y : in  std_logic;
      z : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

begin
   s(8) <= in1(8);
   s(9) <= in1(9);
   u10:ubha port map (in1(10), in2(10), c(11), s(10));
   u11:ubfa port map (in1(11), in2(11), in3(11), c(12), s(11));
   u12:ubfa port map (in1(12), in2(12), in3(12), c(13), s(12));
   u13:ubfa port map (in1(13), in2(13), in3(13), c(14), s(13));
   u14:ubfa port map (in1(14), in2(14), in3(14), c(15), s(14));
   u15:ubfa port map (in1(15), in2(15), in3(15), c(16), s(15));
   u16:ubfa port map (in1(16), in2(16), in3(16), c(17), s(16));
   u17:ubfa port map (in1(17), in2(17), in3(17), c(18), s(17));
   u18:ubfa port map (in1(18), in2(18), in3(18), c(19), s(18));
   u19:ubfa port map (in1(19), in2(19), in3(19), c(20), s(19));
```

```
   u20:ubfa port map (in1(20), in2(20), in3(20), c(21), s(20));
   u21:ubfa port map (in1(21), in2(21), in3(21), c(22), s(21));
   u22:ubfa port map (in1(22), in2(22), in3(22), c(23), s(22));
   u23:ubfa port map (in1(23), in2(23), in3(23), c(24), s(23));
   u24:ubfa port map (in1(24), in2(24), in3(24), c(25), s(24));
   u25:ubfa port map (in1(25), in2(25), in3(25), c(26), s(25));
   s(26) <= in3(26);
end structure;


library ieee;
use ieee.std_logic_1164.all;

entity csa_26_8_26_11_30_15 is port (
   in1   : in  std_logic_vector(26 downto 8);
   in2   : in  std_logic_vector(26 downto 11);
   in3   : in  std_logic_vector(30 downto 15);
   c     : out std_logic_vector(27 downto 12);
   s     : out std_logic_vector(30 downto 8));
end csa_26_8_26_11_30_15;

architecture structure of csa_26_8_26_11_30_15 is
   component ubha port(
      x : in  std_logic;
      y : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

   component ubfa port(
      x : in  std_logic;
      y : in  std_logic;
      z : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

begin
   s(8) <= in1(8);
   s(9) <= in1(9);
   s(10) <= in1(10);
   u11:ubha port map (in1(11), in2(11), c(12), s(11));
   u12:ubha port map (in1(12), in2(12), c(13), s(12));
   u13:ubha port map (in1(13), in2(13), c(14), s(13));
   u14:ubha port map (in1(14), in2(14), c(15), s(14));
   u15:ubfa port map (in1(15), in2(15), in3(15), c(16), s(15));
   u16:ubfa port map (in1(16), in2(16), in3(16), c(17), s(16));
   u17:ubfa port map (in1(17), in2(17), in3(17), c(18), s(17));
   u18:ubfa port map (in1(18), in2(18), in3(18), c(19), s(18));
   u19:ubfa port map (in1(19), in2(19), in3(19), c(20), s(19));
   u20:ubfa port map (in1(20), in2(20), in3(20), c(21), s(20));
   u21:ubfa port map (in1(21), in2(21), in3(21), c(22), s(21));
   u22:ubfa port map (in1(22), in2(22), in3(22), c(23), s(22));
   u23:ubfa port map (in1(23), in2(23), in3(23), c(24), s(23));
   u24:ubfa port map (in1(24), in2(24), in3(24), c(25), s(24));
   u25:ubfa port map (in1(25), in2(25), in3(25), c(26), s(25));
   u26:ubfa port map (in1(26), in2(26), in3(26), c(27), s(26));
```

```
   s(27) <= in3(27);
   s(28) <= in3(28);
   s(29) <= in3(29);
   s(30) <= in3(30);
end structure;


library ieee;
use ieee.std_logic_1164.all;

entity csa_27_12_28_13_29_14 is port (
   in1   : in  std_logic_vector(27 downto 12);
   in2   : in  std_logic_vector(28 downto 13);
   in3   : in  std_logic_vector(29 downto 14);
   c     : out std_logic_vector(29 downto 14);
   s     : out std_logic_vector(29 downto 12));
end csa_27_12_28_13_29_14;

architecture structure of csa_27_12_28_13_29_14 is
   component ubha port(
      x : in  std_logic;
      y : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

   component ubfa port(
      x : in  std_logic;
      y : in  std_logic;
      z : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

begin
   s(12) <= in1(12);
   u13:ubha port map (in1(13), in2(13), c(14), s(13));
   u14:ubfa port map (in1(14), in2(14), in3(14), c(15), s(14));
   u15:ubfa port map (in1(15), in2(15), in3(15), c(16), s(15));
   u16:ubfa port map (in1(16), in2(16), in3(16), c(17), s(16));
   u17:ubfa port map (in1(17), in2(17), in3(17), c(18), s(17));
   u18:ubfa port map (in1(18), in2(18), in3(18), c(19), s(18));
   u19:ubfa port map (in1(19), in2(19), in3(19), c(20), s(19));
   u20:ubfa port map (in1(20), in2(20), in3(20), c(21), s(20));
   u21:ubfa port map (in1(21), in2(21), in3(21), c(22), s(21));
   u22:ubfa port map (in1(22), in2(22), in3(22), c(23), s(22));
   u23:ubfa port map (in1(23), in2(23), in3(23), c(24), s(23));
   u24:ubfa port map (in1(24), in2(24), in3(24), c(25), s(24));
   u25:ubfa port map (in1(25), in2(25), in3(25), c(26), s(25));
   u26:ubfa port map (in1(26), in2(26), in3(26), c(27), s(26));
   u27:ubfa port map (in1(27), in2(27), in3(27), c(28), s(27));
   u28:ubha port map (in2(28), in3(28), c(29), s(28));
   s(29) <= in3(29);
end structure;


library ieee;
```

```
use ieee.std_logic_1164.all;

entity csa_27_12_30_8_30_12 is port (
   in1   : in  std_logic_vector(27 downto 12);
   in2   : in  std_logic_vector(30 downto 8);
   in3   : in  std_logic_vector(30 downto 12);
   c     : out std_logic_vector(31 downto 13);
   s     : out std_logic_vector(30 downto 8));
end csa_27_12_30_8_30_12;

architecture structure of csa_27_12_30_8_30_12 is
   component ubha port(
      x : in  std_logic;
      y : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

   component ubfa port(
      x : in  std_logic;
      y : in  std_logic;
      z : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

begin
   s(8) <= in2(8);
   s(9) <= in2(9);
   s(10) <= in2(10);
   s(11) <= in2(11);
   u12:ubfa port map (in1(12), in2(12), in3(12), c(13), s(12));
   u13:ubfa port map (in1(13), in2(13), in3(13), c(14), s(13));
   u14:ubfa port map (in1(14), in2(14), in3(14), c(15), s(14));
   u15:ubfa port map (in1(15), in2(15), in3(15), c(16), s(15));
   u16:ubfa port map (in1(16), in2(16), in3(16), c(17), s(16));
   u17:ubfa port map (in1(17), in2(17), in3(17), c(18), s(17));
   u18:ubfa port map (in1(18), in2(18), in3(18), c(19), s(18));
   u19:ubfa port map (in1(19), in2(19), in3(19), c(20), s(19));
   u20:ubfa port map (in1(20), in2(20), in3(20), c(21), s(20));
   u21:ubfa port map (in1(21), in2(21), in3(21), c(22), s(21));
   u22:ubfa port map (in1(22), in2(22), in3(22), c(23), s(22));
   u23:ubfa port map (in1(23), in2(23), in3(23), c(24), s(23));
   u24:ubfa port map (in1(24), in2(24), in3(24), c(25), s(24));
   u25:ubfa port map (in1(25), in2(25), in3(25), c(26), s(25));
   u26:ubfa port map (in1(26), in2(26), in3(26), c(27), s(26));
   u27:ubfa port map (in1(27), in2(27), in3(27), c(28), s(27));
   u28:ubha port map (in2(28), in3(28), c(29), s(28));
   u29:ubha port map (in2(29), in3(29), c(30), s(29));
   u30:ubha port map (in2(30), in3(30), c(31), s(30));
end structure;


library ieee;
use ieee.std_logic_1164.all;

entity csa_29_12_29_14_30_15 is port (
```

```
   in1   : in  std_logic_vector(29 downto 12);
   in2   : in  std_logic_vector(29 downto 14);
   in3   : in  std_logic_vector(30 downto 15);
   c     : out std_logic_vector(30 downto 15);
   s     : out std_logic_vector(30 downto 12));
end csa_29_12_29_14_30_15;

architecture structure of csa_29_12_29_14_30_15 is
   component ubha port(
      x : in  std_logic;
      y : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

   component ubfa port(
      x : in  std_logic;
      y : in  std_logic;
      z : in  std_logic;
      c : out std_logic;
      s : out std_logic);
   end component;

begin
   s(12) <= in1(12);
   s(13) <= in1(13);
   u14:ubha port map (in1(14), in2(14), c(15), s(14));
   u15:ubfa port map (in1(15), in2(15), in3(15), c(16), s(15));
   u16:ubfa port map (in1(16), in2(16), in3(16), c(17), s(16));
   u17:ubfa port map (in1(17), in2(17), in3(17), c(18), s(17));
   u18:ubfa port map (in1(18), in2(18), in3(18), c(19), s(18));
   u19:ubfa port map (in1(19), in2(19), in3(19), c(20), s(19));
   u20:ubfa port map (in1(20), in2(20), in3(20), c(21), s(20));
   u21:ubfa port map (in1(21), in2(21), in3(21), c(22), s(21));
   u22:ubfa port map (in1(22), in2(22), in3(22), c(23), s(22));
   u23:ubfa port map (in1(23), in2(23), in3(23), c(24), s(23));
   u24:ubfa port map (in1(24), in2(24), in3(24), c(25), s(24));
   u25:ubfa port map (in1(25), in2(25), in3(25), c(26), s(25));
   u26:ubfa port map (in1(26), in2(26), in3(26), c(27), s(26));
   u27:ubfa port map (in1(27), in2(27), in3(27), c(28), s(27));
   u28:ubfa port map (in1(28), in2(28), in3(28), c(29), s(28));
   u29:ubfa port map (in1(29), in2(29), in3(29), c(30), s(29));
   s(30) <= in3(30);
end structure;


library ieee;
use ieee.std_logic_1164.all;

entity ubha is port (
   x : in  std_logic;
   y : in  std_logic;
   c : out std_logic;
   s : out std_logic);
end ubha;

architecture behav of ubha is
```

```
begin
   c <= x and y;
   s <= x xor y;
end behav;


library ieee;
use ieee.std_logic_1164.all;

entity ubfa is port (
   x : in  std_logic;
   y : in  std_logic;
   z : in  std_logic;
   c : out std_logic;
   s : out std_logic);
end ubfa;

architecture behav of ubfa is
begin
   c <= ( x and y ) or ( y and z ) or ( z and x );
   s <= x xor y xor z;
end behav;


library ieee;
use ieee.std_logic_1164.all;

entity cla is port (
   x     : in  std_logic_vector ( 31 downto 0 );
   y     : in  std_logic_vector ( 31 downto 7 );
   s     : out std_logic_vector ( 31 downto 0 ));
end cla;

architecture structure of cla is
   component gpgenerator port(
      a  : in  std_logic;
      b  : in  std_logic;
      g  : out std_logic;
      p  : out std_logic);
   end component;

   component claunit_25 port (
      g : in  std_logic_vector ( 24 downto 0 );
      p : in  std_logic_vector ( 24 downto 0 );
      c : out std_logic_vector ( 25 downto 1 ));
   end component;

   signal g :  std_logic_vector ( 31 downto 7 );
   signal p :  std_logic_vector ( 31 downto 7 );
   signal c :  std_logic_vector ( 32 downto 8 );

begin
   s(0) <= x(0);
   s(1) <= x(1);
   s(2) <= x(2);
   s(3) <= x(3);
   s(4) <= x(4);
```

```
  s(5) <= x(5);
  s(6) <= x(6);
  s(7) <= p(7);
  s(8) <= c(8) xor p(8);
  s(9) <= c(9) xor p(9);
  s(10) <= c(10) xor p(10);
  s(11) <= c(11) xor p(11);
  s(12) <= c(12) xor p(12);
  s(13) <= c(13) xor p(13);
  s(14) <= c(14) xor p(14);
  s(15) <= c(15) xor p(15);
  s(16) <= c(16) xor p(16);
  s(17) <= c(17) xor p(17);
  s(18) <= c(18) xor p(18);
  s(19) <= c(19) xor p(19);
  s(20) <= c(20) xor p(20);
  s(21) <= c(21) xor p(21);
  s(22) <= c(22) xor p(22);
  s(23) <= c(23) xor p(23);
  s(24) <= c(24) xor p(24);
  s(25) <= c(25) xor p(25);
  s(26) <= c(26) xor p(26);
  s(27) <= c(27) xor p(27);
  s(28) <= c(28) xor p(28);
  s(29) <= c(29) xor p(29);
  s(30) <= c(30) xor p(30);
  s(31) <= c(31) xor p(31);
  u0:gpgenerator port map (x(7), y(7), g(7), p(7));
  u1:gpgenerator port map (x(8), y(8), g(8), p(8));
  u2:gpgenerator port map (x(9), y(9), g(9), p(9));
  u3:gpgenerator port map (x(10), y(10), g(10), p(10));
  u4:gpgenerator port map (x(11), y(11), g(11), p(11));
  u5:gpgenerator port map (x(12), y(12), g(12), p(12));
  u6:gpgenerator port map (x(13), y(13), g(13), p(13));
  u7:gpgenerator port map (x(14), y(14), g(14), p(14));
  u8:gpgenerator port map (x(15), y(15), g(15), p(15));
  u9:gpgenerator port map (x(16), y(16), g(16), p(16));
  u10:gpgenerator port map (x(17), y(17), g(17), p(17));
  u11:gpgenerator port map (x(18), y(18), g(18), p(18));
  u12:gpgenerator port map (x(19), y(19), g(19), p(19));
  u13:gpgenerator port map (x(20), y(20), g(20), p(20));
  u14:gpgenerator port map (x(21), y(21), g(21), p(21));
  u15:gpgenerator port map (x(22), y(22), g(22), p(22));
  u16:gpgenerator port map (x(23), y(23), g(23), p(23));
  u17:gpgenerator port map (x(24), y(24), g(24), p(24));
  u18:gpgenerator port map (x(25), y(25), g(25), p(25));
  u19:gpgenerator port map (x(26), y(26), g(26), p(26));
  u20:gpgenerator port map (x(27), y(27), g(27), p(27));
  u21:gpgenerator port map (x(28), y(28), g(28), p(28));
  u22:gpgenerator port map (x(29), y(29), g(29), p(29));
  u23:gpgenerator port map (x(30), y(30), g(30), p(30));
  u24:gpgenerator port map (x(31), y(31), g(31), p(31));
  u25:claunit_25 port map (g, p, c);
end structure;


library ieee;
```

```
use ieee.std_logic_1164.all;

entity gpgenerator is port (
   a  : in  std_logic;
   b  : in  std_logic;
   g  : out std_logic;
   p  : out std_logic);
end gpgenerator;

architecture behav of gpgenerator is
begin
   g <= a and b;
   p <= a xor b;
end behav;


library ieee;
use ieee.std_logic_1164.all;

entity claunit_25 is port (
   g : in  std_logic_vector ( 24 downto 0 );
   p : in  std_logic_vector ( 24 downto 0 );
   c : out std_logic_vector ( 25 downto 1 ));
end claunit_25;

architecture claunit_25 of claunit_25 is
begin
   c(1) <= g(0);
   c(2) <= g(1) or ( p(1) and g(0) );
   c(3) <= g(2) or ( p(2) and g(1) ) or ( p(2) and p(1) and g(0) );
   c(4) <= g(3) or ( p(3) and g(2) ) or ( p(3) and p(2) and g(1) ) or
      ( p(3) and p(2) and p(1) and g(0) );
   c(5) <= g(4) or ( p(4) and g(3) ) or ( p(4) and p(3) and g(2) ) or
      ( p(4) and p(3) and p(2) and g(1) ) or
      ( p(4) and p(3) and p(2) and p(1) and g(0) );
   c(6) <= g(5) or ( p(5) and g(4) ) or ( p(5) and p(4) and g(3) ) or
      ( p(5) and p(4) and p(3) and g(2) ) or
      ( p(5) and p(4) and p(3) and p(2) and g(1) ) or
      ( p(5) and p(4) and p(3) and p(2) and p(1) and g(0) );
   c(7) <= g(6) or ( p(6) and g(5) ) or ( p(6) and p(5) and g(4) ) or
      ( p(6) and p(5) and p(4) and g(3) ) or
      ( p(6) and p(5) and p(4) and p(3) and g(2) ) or
      ( p(6) and p(5) and p(4) and p(3) and p(2) and g(1) ) or
      ( p(6) and p(5) and p(4) and p(3) and p(2) and p(1) and g(0) );
   c(8) <= g(7) or ( p(7) and g(6) ) or ( p(7) and p(6) and g(5) ) or
      ( p(7) and p(6) and p(5) and g(4) ) or
      ( p(7) and p(6) and p(5) and p(4) and g(3) ) or
      ( p(7) and p(6) and p(5) and p(4) and p(3) and g(2) ) or
      ( p(7) and p(6) and p(5) and p(4) and p(3) and p(2) and g(1) ) or
      ( p(7) and p(6) and p(5) and p(4) and p(3) and p(2) and p(1) and
g(0) );
   c(9) <= g(8) or ( p(8) and g(7) ) or ( p(8) and p(7) and g(6) ) or
      ( p(8) and p(7) and p(6) and g(5) ) or
      ( p(8) and p(7) and p(6) and p(5) and g(4) ) or
      ( p(8) and p(7) and p(6) and p(5) and p(4) and g(3) ) or
      ( p(8) and p(7) and p(6) and p(5) and p(4) and p(3) and g(2) ) or
```

```
      ( p(8) and p(7) and p(6) and p(5) and p(4) and p(3) and p(2) and
g(1) ) or
      ( p(8) and p(7) and p(6) and p(5) and p(4) and p(3) and p(2) and
p(1) and g(0) );
   c(10) <= g(9) or ( p(9) and g(8) ) or ( p(9) and p(8) and g(7) ) or
      ( p(9) and p(8) and p(7) and g(6) ) or
      ( p(9) and p(8) and p(7) and p(6) and g(5) ) or
      ( p(9) and p(8) and p(7) and p(6) and p(5) and g(4) ) or
      ( p(9) and p(8) and p(7) and p(6) and p(5) and p(4) and g(3) ) or
      ( p(9) and p(8) and p(7) and p(6) and p(5) and p(4) and p(3) and
g(2) ) or
      ( p(9) and p(8) and p(7) and p(6) and p(5) and p(4) and p(3) and
p(2) and g(1) ) or
      ( p(9) and p(8) and p(7) and p(6) and p(5) and p(4) and
        p(3) and p(2) and p(1) and g(0) );
   c(11) <= g(10) or ( p(10) and g(9) ) or ( p(10) and p(9) and g(8) )
or
      ( p(10) and p(9) and p(8) and g(7) ) or
      ( p(10) and p(9) and p(8) and p(7) and g(6) ) or
      ( p(10) and p(9) and p(8) and p(7) and p(6) and g(5) ) or
      ( p(10) and p(9) and p(8) and p(7) and p(6) and p(5) and g(4) )
or
      ( p(10) and p(9) and p(8) and p(7) and p(6) and p(5) and p(4) and
g(3) ) or
      ( p(10) and p(9) and p(8) and p(7) and p(6) and p(5) and p(4) and
p(3) and g(2) ) or
      ( p(10) and p(9) and p(8) and p(7) and p(6) and p(5) and
        p(4) and p(3) and p(2) and g(1) ) or
      ( p(10) and p(9) and p(8) and p(7) and p(6) and p(5) and
        p(4) and p(3) and p(2) and p(1) and g(0) );
   c(12) <= g(11) or ( p(11) and g(10) ) or ( p(11) and p(10) and g(9)
) or
      ( p(11) and p(10) and p(9) and g(8) ) or
      ( p(11) and p(10) and p(9) and p(8) and g(7) ) or
      ( p(11) and p(10) and p(9) and p(8) and p(7) and g(6) ) or
      ( p(11) and p(10) and p(9) and p(8) and p(7) and p(6) and g(5) )
or
      ( p(11) and p(10) and p(9) and p(8) and p(7) and p(6) and p(5)
and g(4) ) or
      ( p(11) and p(10) and p(9) and p(8) and p(7) and p(6) and p(5)
and p(4) and g(3) ) or
      ( p(11) and p(10) and p(9) and p(8) and p(7) and p(6) and
        p(5) and p(4) and p(3) and g(2) ) or
      ( p(11) and p(10) and p(9) and p(8) and p(7) and p(6) and
        p(5) and p(4) and p(3) and p(2) and g(1) ) or
      ( p(11) and p(10) and p(9) and p(8) and p(7) and p(6) and
        p(5) and p(4) and p(3) and p(2) and p(1) and g(0) );
   c(13) <= g(12) or ( p(12) and g(11) ) or ( p(12) and p(11) and g(10)
) or
      ( p(12) and p(11) and p(10) and g(9) ) or
      ( p(12) and p(11) and p(10) and p(9) and g(8) ) or
      ( p(12) and p(11) and p(10) and p(9) and p(8) and g(7) ) or
      ( p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and g(6) )
or
      ( p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and p(6)
and g(5) ) or
      ( p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and
```

```
        p(6) and p(5) and g(4) ) or
      ( p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and
        p(6) and p(5) and p(4) and g(3) ) or
      ( p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and
        p(6) and p(5) and p(4) and p(3) and g(2) ) or
      ( p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and
        p(6) and p(5) and p(4) and p(3) and p(2) and g(1) ) or
      ( p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and p(6)
and p(5) and p(4) and p(3) and p(2) and p(1) and g(0) );
   c(14) <= g(13) or ( p(13) and g(12) ) or ( p(13) and p(12) and g(11)
) or
      ( p(13) and p(12) and p(11) and g(10) ) or
      ( p(13) and p(12) and p(11) and p(10) and g(9) ) or
      ( p(13) and p(12) and p(11) and p(10) and p(9) and g(8) ) or
      ( p(13) and p(12) and p(11) and p(10) and p(9) and p(8) and g(7)
) or
      ( p(13) and p(12) and p(11) and p(10) and p(9) and p(8) and p(7)
and g(6) ) or
      ( p(13) and p(12) and p(11) and p(10) and p(9) and p(8) and
        p(7) and p(6) and g(5) ) or
      ( p(13) and p(12) and p(11) and p(10) and p(9) and p(8) and
        p(7) and p(6) and p(5) and g(4) ) or
      ( p(13) and p(12) and p(11) and p(10) and p(9) and p(8) and
        p(7) and p(6) and p(5) and p(4) and g(3) ) or
      ( p(13) and p(12) and p(11) and p(10) and p(9) and p(8) and
        p(7) and p(6) and p(5) and p(4) and p(3) and g(2) ) or
      ( p(13) and p(12) and p(11) and p(10) and p(9) and p(8) and
        p(7) and p(6) and p(5) and p(4) and p(3) and p(2) and g(1) ) or
      ( p(13) and p(12) and p(11) and p(10) and p(9) and p(8) and
        p(7) and p(6) and p(5) and p(4) and p(3) and p(2) and p(1) and
g(0) );
   c(15) <= g(14) or ( p(14) and g(13) ) or ( p(14) and p(13) and g(12)
) or
      ( p(14) and p(13) and p(12) and g(11) ) or
      ( p(14) and p(13) and p(12) and p(11) and g(10) ) or
      ( p(14) and p(13) and p(12) and p(11) and p(10) and g(9) ) or
      ( p(14) and p(13) and p(12) and p(11) and p(10) and p(9) and g(8)
) or
      ( p(14) and p(13) and p(12) and p(11) and p(10) and p(9) and
        p(8) and g(7) ) or
      ( p(14) and p(13) and p(12) and p(11) and p(10) and p(9) and
        p(8) and p(7) and g(6) ) or
      ( p(14) and p(13) and p(12) and p(11) and p(10) and p(9) and
        p(8) and p(7) and p(6) and g(5) ) or
      ( p(14) and p(13) and p(12) and p(11) and p(10) and p(9) and
        p(8) and p(7) and p(6) and p(5) and g(4) ) or
      ( p(14) and p(13) and p(12) and p(11) and p(10) and p(9) and
        p(8) and p(7) and p(6) and p(5) and p(4) and g(3) ) or
      ( p(14) and p(13) and p(12) and p(11) and p(10) and p(9) and
        p(8) and p(7) and p(6) and p(5) and p(4) and p(3) and g(2) ) or
      ( p(14) and p(13) and p(12) and p(11) and p(10) and p(9) and
        p(8) and p(7) and p(6) and p(5) and p(4) and p(3) and p(2) and
g(1) ) or
      ( p(14) and p(13) and p(12) and p(11) and p(10) and p(9) and
        p(8) and p(7) and p(6) and p(5) and p(4) and p(3) and p(2) and
p(1) and g(0) );
```

```
   c(16) <= g(15) or ( p(15) and g(14) ) or ( p(15) and p(14) and g(13)
) or
       ( p(15) and p(14) and p(13) and g(12) ) or
       ( p(15) and p(14) and p(13) and p(12) and g(11) ) or
       ( p(15) and p(14) and p(13) and p(12) and p(11) and g(10) ) or
       ( p(15) and p(14) and p(13) and p(12) and p(11) and p(10) and
g(9) ) or
       ( p(15) and p(14) and p(13) and p(12) and p(11) and p(10) and
p(9) and g(8) ) or
       ( p(15) and p(14) and p(13) and p(12) and p(11) and p(10) and
         p(9) and p(8) and g(7) ) or
       ( p(15) and p(14) and p(13) and p(12) and p(11) and p(10) and
         p(9) and p(8) and p(7) and g(6) ) or
       ( p(15) and p(14) and p(13) and p(12) and p(11) and p(10) and
         p(9) and p(8) and p(7) and p(6) and g(5) ) or
       ( p(15) and p(14) and p(13) and p(12) and p(11) and p(10) and
         p(9) and p(8) and p(7) and p(6) and p(5) and g(4) ) or
       ( p(15) and p(14) and p(13) and p(12) and p(11) and p(10) and
         p(9) and p(8) and p(7) and p(6) and p(5) and p(4) and g(3) ) or
       ( p(15) and p(14) and p(13) and p(12) and p(11) and p(10) and
         p(9) and p(8) and p(7) and p(6) and p(5) and p(4) and p(3) and
g(2) ) or
       ( p(15) and p(14) and p(13) and p(12) and p(11) and p(10) and
         p(9) and p(8) and p(7) and p(6) and p(5) and p(4) and p(3) and
p(2) and g(1) ) or
       ( p(15) and p(14) and p(13) and p(12) and p(11) and p(10) and
         p(9) and p(8) and p(7) and p(6) and p(5) and p(4) and p(3) and
         p(2) and p(1) and g(0) );
   c(17) <= g(16) or ( p(16) and g(15) ) or ( p(16) and p(15) and g(14)
) or
       ( p(16) and p(15) and p(14) and g(13) ) or
       ( p(16) and p(15) and p(14) and p(13) and g(12) ) or
       ( p(16) and p(15) and p(14) and p(13) and p(12) and g(11) ) or
       ( p(16) and p(15) and p(14) and p(13) and p(12) and p(11) and
g(10) ) or
       ( p(16) and p(15) and p(14) and p(13) and p(12) and p(11) and
p(10) and g(9) ) or
       ( p(16) and p(15) and p(14) and p(13) and p(12) and p(11) and
         p(10) and p(9) and g(8) ) or ( p(16) and p(15) and p(14) and
         p(13) and p(12) and p(11) and p(10) and p(9) and p(8) and g(7)
) or
       ( p(16) and p(15) and p(14) and p(13) and p(12) and p(11) and
         p(10) and p(9) and p(8) and p(7) and g(6) ) or
       ( p(16) and p(15) and p(14) and p(13) and p(12) and p(11) and
         p(10) and p(9) and p(8) and p(7) and p(6) and g(5) ) or
       ( p(16) and p(15) and p(14) and p(13) and p(12) and p(11) and
         p(10) and p(9) and p(8) and p(7) and p(6) and p(5) and g(4) )
or
       ( p(16) and p(15) and p(14) and p(13) and p(12) and p(11) and
         p(10) and p(9) and p(8) and p(7) and p(6) and p(5) and p(4) and
g(3) ) or
       ( p(16) and p(15) and p(14) and p(13) and p(12) and p(11) and
         p(10) and p(9) and p(8) and p(7) and p(6) and p(5) and p(4) and
         p(3) and g(2) ) or
       ( p(16) and p(15) and p(14) and p(13) and p(12) and p(11) and
         p(10) and p(9) and p(8) and p(7) and p(6) and p(5) and p(4) and
         p(3) and p(2) and g(1) ) or
```

```
      ( p(16) and p(15) and p(14) and p(13) and p(12) and p(11) and
        p(10) and p(9) and p(8) and p(7) and p(6) and p(5) and p(4) and
        p(3) and p(2) and p(1) and g(0) );
   c(18) <= g(17) or ( p(17) and g(16) ) or ( p(17) and p(16) and g(15)
) or
      ( p(17) and p(16) and p(15) and g(14) ) or
      ( p(17) and p(16) and p(15) and p(14) and g(13) ) or
      ( p(17) and p(16) and p(15) and p(14) and p(13) and g(12) ) or
      ( p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
g(11) ) or
      ( p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
p(11) and g(10) ) or
      ( p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
        p(11) and p(10) and g(9) ) or
      ( p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
        p(11) and p(10) and p(9) and g(8) ) or
      ( p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
        p(11) and p(10) and p(9) and p(8) and g(7) ) or
      ( p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
        p(11) and p(10) and p(9) and p(8) and p(7) and g(6) ) or
      ( p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
        p(11) and p(10) and p(9) and p(8) and p(7) and p(6) and g(5) )
or
      ( p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
        p(11) and p(10) and p(9) and p(8) and p(7) and p(6) and p(5)
and g(4) ) or
      ( p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
        p(11) and p(10) and p(9) and p(8) and p(7) and p(6) and p(5)
and
        p(4) and g(3) ) or
      ( p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
        p(11) and p(10) and p(9) and p(8) and p(7) and p(6) and p(5)
and
        p(4) and p(3) and g(2) ) or
      ( p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
        p(11) and p(10) and p(9) and p(8) and p(7) and p(6) and p(5)
and
        p(4) and p(3) and p(2) and g(1) ) or
      ( p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
        p(11) and p(10) and p(9) and p(8) and p(7) and p(6) and p(5)
and
        p(4) and p(3) and p(2) and p(1) and g(0) );
   c(19) <= g(18) or ( p(18) and g(17) ) or ( p(18) and p(17) and g(16)
) or
      ( p(18) and p(17) and p(16) and g(15) ) or
      ( p(18) and p(17) and p(16) and p(15) and g(14) ) or
      ( p(18) and p(17) and p(16) and p(15) and p(14) and g(13) ) or
      ( p(18) and p(17) and p(16) and p(15) and p(14) and p(13) and
g(12) ) or
      ( p(18) and p(17) and p(16) and p(15) and p(14) and p(13) and
p(12) and g(11) ) or
      ( p(18) and p(17) and p(16) and p(15) and p(14) and p(13) and
        p(12) and p(11) and g(10) ) or
      ( p(18) and p(17) and p(16) and p(15) and p(14) and p(13) and
        p(12) and p(11) and p(10) and g(9) ) or
      ( p(18) and p(17) and p(16) and p(15) and p(14) and p(13) and
        p(12) and p(11) and p(10) and p(9) and g(8) ) or
```

```
      ( p(18) and p(17) and p(16) and p(15) and p(14) and p(13) and
        p(12) and p(11) and p(10) and p(9) and p(8) and g(7) ) or
      ( p(18) and p(17) and p(16) and p(15) and p(14) and p(13) and
        p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and g(6) )
or
      ( p(18) and p(17) and p(16) and p(15) and p(14) and p(13) and
        p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and p(6)
and g(5) ) or
      ( p(18) and p(17) and p(16) and p(15) and p(14) and p(13) and
        p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and
        p(6) and p(5) and g(4) ) or
      ( p(18) and p(17) and p(16) and p(15) and p(14) and p(13) and
        p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and p(6)
and
        p(5) and p(4) and g(3) ) or
      ( p(18) and p(17) and p(16) and p(15) and p(14) and p(13) and
        p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and p(6)
        and p(5) and p(4) and p(3) and g(2) ) or
      ( p(18) and p(17) and p(16) and p(15) and p(14) and p(13) and
        p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and p(6)
and
        p(5) and p(4) and p(3) and p(2) and g(1) ) or
      ( p(18) and p(17) and p(16) and p(15) and p(14) and p(13) and
        p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and p(6)
and
        p(5) and p(4) and p(3) and p(2) and p(1) and g(0) );
   c(20) <= g(19) or ( p(19) and g(18) ) or ( p(19) and p(18) and g(17)
) or
      ( p(19) and p(18) and p(17) and g(16) ) or
      ( p(19) and p(18) and p(17) and p(16) and g(15) ) or
      ( p(19) and p(18) and p(17) and p(16) and p(15) and g(14) ) or
      ( p(19) and p(18) and p(17) and p(16) and p(15) and p(14) and
g(13) ) or
      ( p(19) and p(18) and p(17) and p(16) and p(15) and p(14) and
p(13) and g(12) ) or
      ( p(19) and p(18) and p(17) and p(16) and p(15) and p(14) and
p(13) and
        p(12) and g(11) ) or
      ( p(19) and p(18) and p(17) and p(16) and p(15) and p(14) and
p(13) and
        p(12) and p(11) and g(10) ) or
      ( p(19) and p(18) and p(17) and p(16) and p(15) and p(14) and
p(13) and
        p(12) and p(11) and p(10) and g(9) ) or
      ( p(19) and p(18) and p(17) and p(16) and p(15) and p(14) and
p(13) and
        p(12) and p(11) and p(10) and p(9) and g(8) ) or
      ( p(19) and p(18) and p(17) and p(16) and p(15) and p(14) and
p(13) and
        p(12) and p(11) and p(10) and p(9) and p(8) and g(7) ) or
      ( p(19) and p(18) and p(17) and p(16) and p(15) and p(14) and
p(13) and
        p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and g(6) )
or
      ( p(19) and p(18) and p(17) and p(16) and p(15) and p(14) and
p(13) and
```

```
        p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and p(6)
and g(5) ) or
      ( p(19) and p(18) and p(17) and p(16) and p(15) and p(14) and
        p(13) and p(12) and p(11) and p(10) and p(9) and p(8) and p(7)
and
        p(6) and p(5) and g(4) ) or
      ( p(19) and p(18) and p(17) and p(16) and p(15) and p(14) and
p(13) and
        p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and p(6)
and
        p(5) and p(4) and g(3) ) or
      ( p(19) and p(18) and p(17) and p(16) and p(15) and p(14) and
p(13) and
        p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and p(6)
and p(5) and
        p(4) and p(3) and g(2) ) or
      ( p(19) and p(18) and p(17) and p(16) and p(15) and p(14) and
p(13) and
        p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and p(6)
and p(5) and
        p(4) and p(3) and p(2) and g(1) ) or
      ( p(19) and p(18) and p(17) and p(16) and p(15) and p(14) and
p(13) and
        p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and p(6)
and p(5) and
        p(4) and p(3) and p(2) and p(1) and g(0) );
   c(21) <= g(20) or ( p(20) and g(19) ) or ( p(20) and p(19) and g(18)
) or
      ( p(20) and p(19) and p(18) and g(17) ) or
      ( p(20) and p(19) and p(18) and p(17) and g(16) ) or
      ( p(20) and p(19) and p(18) and p(17) and p(16) and g(15) ) or
      ( p(20) and p(19) and p(18) and p(17) and p(16) and p(15) and
g(14) ) or
      ( p(20) and p(19) and p(18) and p(17) and p(16) and p(15) and
p(14) and g(13) ) or
      ( p(20) and p(19) and p(18) and p(17) and p(16) and p(15) and
        p(14) and p(13) and g(12) ) or
      ( p(20) and p(19) and p(18) and p(17) and p(16) and p(15) and
        p(14) and p(13) and p(12) and g(11) ) or
      ( p(20) and p(19) and p(18) and p(17) and p(16) and p(15) and
        p(14) and p(13) and p(12) and p(11) and g(10) ) or
      ( p(20) and p(19) and p(18) and p(17) and p(16) and p(15) and
        p(14) and p(13) and p(12) and p(11) and p(10) and g(9) ) or
      ( p(20) and p(19) and p(18) and p(17) and p(16) and p(15) and
        p(14) and p(13) and p(12) and p(11) and p(10) and p(9) and g(8)
) or
      ( p(20) and p(19) and p(18) and p(17) and p(16) and p(15) and
p(14) and
        p(13) and p(12) and p(11) and p(10) and p(9) and p(8) and g(7)
) or
      ( p(20) and p(19) and p(18) and p(17) and p(16) and p(15) and
p(14) and
        p(13) and p(12) and p(11) and p(10) and p(9) and p(8) and p(7)
and g(6) ) or
      ( p(20) and p(19) and p(18) and p(17) and p(16) and p(15) and
        p(14) and p(13) and p(12) and p(11) and p(10) and p(9) and p(8)
and
```

```
        p(7) and p(6) and g(5) ) or
      ( p(20) and p(19) and p(18) and p(17) and p(16) and p(15) and
p(14) and
        p(13) and p(12) and p(11) and p(10) and p(9) and p(8) and p(7)
and
        p(6) and p(5) and g(4) ) or
      ( p(20) and p(19) and p(18) and p(17) and p(16) and p(15) and
p(14) and
        p(13) and p(12) and p(11) and p(10) and p(9) and p(8) and p(7)
and
        p(6) and p(5) and p(4) and g(3) ) or
      ( p(20) and p(19) and p(18) and p(17) and p(16) and p(15) and
p(14) and
        p(13) and p(12) and p(11) and p(10) and p(9) and p(8) and p(7)
and
        p(6) and p(5) and p(4) and p(3) and g(2) ) or
      ( p(20) and p(19) and p(18) and p(17) and p(16) and p(15) and
p(14) and
        p(13) and p(12) and p(11) and p(10) and p(9) and p(8) and p(7)
and
        p(6) and p(5) and p(4) and p(3) and p(2) and g(1) ) or
      ( p(20) and p(19) and p(18) and p(17) and p(16) and p(15) and
p(14) and
        p(13) and p(12) and p(11) and p(10) and p(9) and p(8) and p(7)
and
        p(6) and p(5) and p(4) and p(3) and p(2) and p(1) and g(0) );
   c(22) <= g(21) or ( p(21) and g(20) ) or ( p(21) and p(20) and g(19)
) or
      ( p(21) and p(20) and p(19) and g(18) ) or
      ( p(21) and p(20) and p(19) and p(18) and g(17) ) or
      ( p(21) and p(20) and p(19) and p(18) and p(17) and g(16) ) or
      ( p(21) and p(20) and p(19) and p(18) and p(17) and p(16) and
g(15) ) or
      ( p(21) and p(20) and p(19) and p(18) and p(17) and p(16) and
p(15) and g(14) ) or
      ( p(21) and p(20) and p(19) and p(18) and p(17) and p(16) and
        p(15) and p(14) and g(13) ) or
      ( p(21) and p(20) and p(19) and p(18) and p(17) and p(16) and
        p(15) and p(14) and p(13) and g(12) ) or
      ( p(21) and p(20) and p(19) and p(18) and p(17) and p(16) and
        p(15) and p(14) and p(13) and p(12) and g(11) ) or
      ( p(21) and p(20) and p(19) and p(18) and p(17) and p(16) and
        p(15) and p(14) and p(13) and p(12) and p(11) and g(10) ) or
      ( p(21) and p(20) and p(19) and p(18) and p(17) and p(16) and
        p(15) and p(14) and p(13) and p(12) and p(11) and p(10) and
g(9) ) or
      ( p(21) and p(20) and p(19) and p(18) and p(17) and p(16) and
p(15) and
        p(14) and p(13) and p(12) and p(11) and p(10) and p(9) and g(8)
) or
      ( p(21) and p(20) and p(19) and p(18) and p(17) and p(16) and
p(15) and
        p(14) and p(13) and p(12) and p(11) and p(10) and p(9) and p(8)
and g(7) ) or
      ( p(21) and p(20) and p(19) and p(18) and p(17) and p(16) and
        p(15) and p(14) and p(13) and p(12) and p(11) and p(10) and
        p(9) and p(8) and p(7) and g(6) ) or
```

```
      ( p(21) and p(20) and p(19) and p(18) and p(17) and p(16) and
        p(15) and p(14) and p(13) and p(12) and p(11) and p(10) and
        p(9) and p(8) and p(7) and p(6) and g(5) ) or
      ( p(21) and p(20) and p(19) and p(18) and p(17) and p(16) and
        p(15) and p(14) and p(13) and p(12) and p(11) and p(10) and
        p(9) and p(8) and p(7) and p(6) and p(5) and g(4) ) or
      ( p(21) and p(20) and p(19) and p(18) and p(17) and p(16) and
        p(15) and p(14) and p(13) and p(12) and p(11) and p(10) and
        p(9) and p(8) and p(7) and p(6) and p(5) and p(4) and g(3) ) or
      ( p(21) and p(20) and p(19) and p(18) and p(17) and p(16) and
p(15) and
        p(14) and p(13) and p(12) and p(11) and p(10) and p(9) and p(8)
and
        p(7) and p(6) and p(5) and p(4) and p(3) and g(2) ) or
      ( p(21) and p(20) and p(19) and p(18) and p(17) and p(16) and
p(15) and
        p(14) and p(13) and p(12) and p(11) and p(10) and p(9) and p(8)
and
        p(7) and p(6) and p(5) and p(4) and p(3) and p(2) and g(1) ) or
      ( p(21) and p(20) and p(19) and p(18) and p(17) and p(16) and
p(15) and
        p(14) and p(13) and p(12) and p(11) and p(10) and p(9) and p(8)
and
        p(7) and p(6) and p(5) and p(4) and p(3) and p(2) and p(1) and
g(0) );
   c(23) <= g(22) or ( p(22) and g(21) ) or ( p(22) and p(21) and g(20)
) or
      ( p(22) and p(21) and p(20) and g(19) ) or
      ( p(22) and p(21) and p(20) and p(19) and g(18) ) or
      ( p(22) and p(21) and p(20) and p(19) and p(18) and g(17) ) or
      ( p(22) and p(21) and p(20) and p(19) and p(18) and p(17) and
g(16) ) or
      ( p(22) and p(21) and p(20) and p(19) and p(18) and p(17) and
p(16) and g(15) ) or
      ( p(22) and p(21) and p(20) and p(19) and p(18) and p(17) and
        p(16) and p(15) and g(14) ) or
      ( p(22) and p(21) and p(20) and p(19) and p(18) and p(17) and
        p(16) and p(15) and p(14) and g(13) ) or
      ( p(22) and p(21) and p(20) and p(19) and p(18) and p(17) and
        p(16) and p(15) and p(14) and p(13) and g(12) ) or
      ( p(22) and p(21) and p(20) and p(19) and p(18) and p(17) and
        p(16) and p(15) and p(14) and p(13) and p(12) and g(11) ) or
      ( p(22) and p(21) and p(20) and p(19) and p(18) and p(17) and
        p(16) and p(15) and p(14) and p(13) and p(12) and p(11) and
g(10) ) or
      ( p(22) and p(21) and p(20) and p(19) and p(18) and p(17) and
p(16) and
        p(15) and p(14) and p(13) and p(12) and p(11) and p(10) and
g(9) ) or
      ( p(22) and p(21) and p(20) and p(19) and p(18) and p(17) and
p(16) and
        p(15) and p(14) and p(13) and p(12) and p(11) and p(10) and
p(9) and g(8) ) or
      ( p(22) and p(21) and p(20) and p(19) and p(18) and p(17) and
        p(16) and p(15) and p(14) and p(13) and p(12) and p(11) and
        p(10) and p(9) and p(8) and g(7) ) or
      ( p(22) and p(21) and p(20) and p(19) and p(18) and p(17) and
```

```
        p(16) and p(15) and p(14) and p(13) and p(12) and p(11) and
        p(10) and p(9) and p(8) and p(7) and g(6) ) or
      ( p(22) and p(21) and p(20) and p(19) and p(18) and p(17) and
        p(16) and p(15) and p(14) and p(13) and p(12) and p(11) and
        p(10) and p(9) and p(8) and p(7) and p(6) and g(5) ) or
      ( p(22) and p(21) and p(20) and p(19) and p(18) and p(17) and
        p(16) and p(15) and p(14) and p(13) and p(12) and p(11) and
        p(10) and p(9) and p(8) and p(7) and p(6) and p(5) and g(4) )
or
      ( p(22) and p(21) and p(20) and p(19) and p(18) and p(17) and
p(16) and
        p(15) and p(14) and p(13) and p(12) and p(11) and p(10) and
p(9) and
        p(8) and p(7) and p(6) and p(5) and p(4) and g(3) ) or
      ( p(22) and p(21) and p(20) and p(19) and p(18) and p(17) and
p(16) and
        p(15) and p(14) and p(13) and p(12) and p(11) and p(10) and
p(9) and
        p(8) and p(7) and p(6) and p(5) and p(4) and p(3) and g(2) ) or
      ( p(22) and p(21) and p(20) and p(19) and p(18) and p(17) and
p(16) and
        p(15) and p(14) and p(13) and p(12) and p(11) and p(10) and
p(9) and
        p(8) and p(7) and p(6) and p(5) and p(4) and p(3) and p(2) and
g(1) ) or
      ( p(22) and p(21) and p(20) and p(19) and p(18) and p(17) and
        p(16) and p(15) and p(14) and p(13) and p(12) and p(11) and
        p(10) and p(9) and p(8) and p(7) and p(6) and p(5) and p(4) and
        p(3) and p(2) and p(1) and g(0) );
   c(24) <= g(23) or ( p(23) and g(22) ) or ( p(23) and p(22) and g(21)
) or
      ( p(23) and p(22) and p(21) and g(20) ) or
      ( p(23) and p(22) and p(21) and p(20) and g(19) ) or
      ( p(23) and p(22) and p(21) and p(20) and p(19) and g(18) ) or
      ( p(23) and p(22) and p(21) and p(20) and p(19) and p(18) and
g(17) ) or
      ( p(23) and p(22) and p(21) and p(20) and p(19) and p(18) and
p(17) and g(16) ) or
      ( p(23) and p(22) and p(21) and p(20) and p(19) and p(18) and
        p(17) and p(16) and g(15) ) or
      ( p(23) and p(22) and p(21) and p(20) and p(19) and p(18) and
        p(17) and p(16) and p(15) and g(14) ) or
      ( p(23) and p(22) and p(21) and p(20) and p(19) and p(18) and
        p(17) and p(16) and p(15) and p(14) and g(13) ) or
      ( p(23) and p(22) and p(21) and p(20) and p(19) and p(18) and
        p(17) and p(16) and p(15) and p(14) and p(13) and g(12) ) or
      ( p(23) and p(22) and p(21) and p(20) and p(19) and p(18) and
        p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
g(11) ) or
      ( p(23) and p(22) and p(21) and p(20) and p(19) and p(18) and
p(17) and
        p(16) and p(15) and p(14) and p(13) and p(12) and p(11) and
g(10) ) or
      ( p(23) and p(22) and p(21) and p(20) and p(19) and p(18) and
        p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
        p(11) and p(10) and g(9) ) or
      ( p(23) and p(22) and p(21) and p(20) and p(19) and p(18) and
```

```
        p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
        p(11) and p(10) and p(9) and g(8) ) or
      ( p(23) and p(22) and p(21) and p(20) and p(19) and p(18) and
        p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
        p(11) and p(10) and p(9) and p(8) and g(7) ) or
      ( p(23) and p(22) and p(21) and p(20) and p(19) and p(18) and
        p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
        p(11) and p(10) and p(9) and p(8) and p(7) and g(6) ) or
      ( p(23) and p(22) and p(21) and p(20) and p(19) and p(18) and
        p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
        p(11) and p(10) and p(9) and p(8) and p(7) and p(6) and g(5) )
or
      ( p(23) and p(22) and p(21) and p(20) and p(19) and p(18) and
p(17) and
        p(16) and p(15) and p(14) and p(13) and p(12) and p(11) and
p(10) and
        p(9) and p(8) and p(7) and p(6) and p(5) and g(4) ) or
      ( p(23) and p(22) and p(21) and p(20) and p(19) and p(18) and
p(17) and
        p(16) and p(15) and p(14) and p(13) and p(12) and p(11) and
p(10) and
        p(9) and p(8) and p(7) and p(6) and p(5) and p(4) and g(3) ) or
      ( p(23) and p(22) and p(21) and p(20) and p(19) and p(18) and
p(17) and
        p(16) and p(15) and p(14) and p(13) and p(12) and p(11) and
p(10) and
        p(9) and p(8) and p(7) and p(6) and p(5) and p(4) and p(3) and
g(2) ) or
      ( p(23) and p(22) and p(21) and p(20) and p(19) and p(18) and
        p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
        p(11) and p(10) and p(9) and p(8) and p(7) and p(6) and p(5)
and
        p(4) and p(3) and p(2) and g(1) ) or
      ( p(23) and p(22) and p(21) and p(20) and p(19) and p(18) and
        p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
        p(11) and p(10) and p(9) and p(8) and p(7) and p(6) and p(5)
and
        p(4) and p(3) and p(2) and p(1) and g(0) );
   c(25) <= g(24) or ( p(24) and g(23) ) or ( p(24) and p(23) and g(22)
) or
      ( p(24) and p(23) and p(22) and g(21) ) or
      ( p(24) and p(23) and p(22) and p(21) and g(20) ) or
      ( p(24) and p(23) and p(22) and p(21) and p(20) and g(19) ) or
      ( p(24) and p(23) and p(22) and p(21) and p(20) and p(19) and
g(18) ) or
      ( p(24) and p(23) and p(22) and p(21) and p(20) and p(19) and
p(18) and g(17) ) or
      ( p(24) and p(23) and p(22) and p(21) and p(20) and p(19) and
        p(18) and p(17) and g(16) ) or
      ( p(24) and p(23) and p(22) and p(21) and p(20) and p(19) and
        p(18) and p(17) and p(16) and g(15) ) or
      ( p(24) and p(23) and p(22) and p(21) and p(20) and p(19) and
        p(18) and p(17) and p(16) and p(15) and g(14) ) or
      ( p(24) and p(23) and p(22) and p(21) and p(20) and p(19) and
        p(18) and p(17) and p(16) and p(15) and p(14) and g(13) ) or
      ( p(24) and p(23) and p(22) and p(21) and p(20) and p(19) and
```

```
        p(18) and p(17) and p(16) and p(15) and p(14) and p(13) and
g(12) ) or
      ( p(24) and p(23) and p(22) and p(21) and p(20) and p(19) and
p(18) and
        p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
g(11) ) or
      ( p(24) and p(23) and p(22) and p(21) and p(20) and p(19) and
        p(18) and p(17) and p(16) and p(15) and p(14) and p(13) and
        p(12) and p(11) and g(10) ) or
      ( p(24) and p(23) and p(22) and p(21) and p(20) and p(19) and
        p(18) and p(17) and p(16) and p(15) and p(14) and p(13) and
        p(12) and p(11) and p(10) and g(9) ) or
      ( p(24) and p(23) and p(22) and p(21) and p(20) and p(19) and
        p(18) and p(17) and p(16) and p(15) and p(14) and p(13) and
        p(12) and p(11) and p(10) and p(9) and g(8) ) or
      ( p(24) and p(23) and p(22) and p(21) and p(20) and p(19) and
        p(18) and p(17) and p(16) and p(15) and p(14) and p(13) and
        p(12) and p(11) and p(10) and p(9) and p(8) and g(7) ) or
      ( p(24) and p(23) and p(22) and p(21) and p(20) and p(19) and
        p(18) and p(17) and p(16) and p(15) and p(14) and p(13) and
        p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and g(6) )
or
      ( p(24) and p(23) and p(22) and p(21) and p(20) and p(19) and
p(18) and
        p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
p(11) and
        p(10) and p(9) and p(8) and p(7) and p(6) and g(5) ) or
      ( p(24) and p(23) and p(22) and p(21) and p(20) and p(19) and
p(18) and
        p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
p(11) and
        p(10) and p(9) and p(8) and p(7) and p(6) and p(5) and g(4) )
or
      ( p(24) and p(23) and p(22) and p(21) and p(20) and p(19) and
p(18) and
        p(17) and p(16) and p(15) and p(14) and p(13) and p(12) and
p(11) and
        p(10) and p(9) and p(8) and p(7) and p(6) and p(5) and p(4) and
g(3) ) or
      ( p(24) and p(23) and p(22) and p(21) and p(20) and p(19) and
        p(18) and p(17) and p(16) and p(15) and p(14) and p(13) and
        p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and
        p(6) and p(5) and p(4) and p(3) and g(2) ) or
      ( p(24) and p(23) and p(22) and p(21) and p(20) and p(19) and
        p(18) and p(17) and p(16) and p(15) and p(14) and p(13) and
        p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and
        p(6) and p(5) and p(4) and p(3) and p(2) and g(1) ) or
      ( p(24) and p(23) and p(22) and p(21) and p(20) and p(19) and
        p(18) and p(17) and p(16) and p(15) and p(14) and p(13) and
        p(12) and p(11) and p(10) and p(9) and p(8) and p(7) and p(6)
and
        p(5) and p(4) and p(3) and p(2) and p(1) and g(0) );
end claunit_25;
```

REFERENCE LIST

[1] S. Chou, "Integration and innovation in the nanoelectronics era," *IEEE International Solid-State Circuits Conference*, vol. 1, pp. 36-41, Feb. 2005.

[2] J.B. Kuo, *CMOS Digital IC*. New York, NY: McGraw-Hill, 1996.

[3] J.M. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits, 2nd Ed.* New Jersey, NJ: Prentice Hall, 2003.

[4] D.A. Neamen, *Semiconductor Physics and Devices, 3rd Ed.* New York, NY: McGraw-Hill, 2003.

[5] G. Moore, "No exponential is forever: but 'forever can be delayed," *IEEE International Solid-State Circuits Conference*, vol. 1, pp. 20-23, Feb. 2003.

[6] T. Sakurai, "Perspectives on power-aware electronics," *IEEE International Solid-State Circuits Conference*, vol. 1, pp. 26-29, Feb. 2003.

[7] S. Mutoh, T. Douseki, Y. Matsuya, T. Aoki, S. Shigematsu, and J. Yamada, "1-V power supply high-speed digital circuit technology with multithreshold-voltage CMOS," *IEEE Journal of Solid-State Circuits*, vol. 30, pp. 847-853, Aug. 1995.

[8] J. Kao, S. Narendra, and A. Chandrakasan, "Subthreshold leakage modeling and reduction techniques," *IEEE International Conference on Computer Aided Design*, pp. 141-148, Nov. 2002.

[9] J. Kao, and A. Chandrakasan, "Dual-threshold voltage techniques for low-power digital circuits," *IEEE Journal of Solid-State Circuits*, vol. 35, pp. 1009-1018, Jul. 2000.

[10] R. Rao, K. Agarwal, D. Sylvester, R. Brown, K. Nowka, and S. Nassif, "Approaches to run-time and standby mode leakage reduction in global buses," *International Symposium on Low Power Electronics and Design*, pp. 188-193, Aug. 2004.

[11] C. Gopalakrishman, and S. Katkoori, "Behavioral synthesis of datapaths with low leakage power," *IEEE International Symposium on Circuits and Systems*, vol. 4, pp. 699-702, May 2002.

[12] M. Anis, S, Areibi, M. Mahmoud, and M. Elmasry, "Dynamic and leakage power reduction in MTCMOS circuits using an automated efficient gate clustering technique," *Design Automation Conference*, pp. 480-485, Jun. 2002.

[13] T. Chang, T. Hwang, and S. Hsu, "Functionality directed clustering for low power MTCMOS design," *Asian and South Pacific Design Automation Conference*, vol. 2, pp. 862-867, Jan. 2005.

[14] V. Khandelwal, and A. Srivastava, "Leakage control through fine-grained placement and sizing of sleep transistors," *IEEE International Conference on Computer Aided Design*, pp. 533-536, Nov. 2004.

[15] J.T. Kao, A.P. Chandrakasan, "Dual-threshold voltage techniques for low-power digital circuits," *IEEE Journal of Solid-State Circuits*, vol. 35, pp. 1009-1018, Jul. 2000.

[16] J. Kao, and A. Chandrakasan, "MTCMOS sequential circuits," *European Solid-State Circuits Conference*, pp. 317-320, Sep. 2001.

[17] U. Ko, A. Pua, A. Hill, and P. Srivastava, "Hybrid dual-threshold design techniques for high-performance processors with low-power features," *International Symposium on Low Power Electronics and Design*, pp. 18-20, Aug. 1997.

[18] B.H., Calhoun, F.A., Honore, and A. Chandrakasan, "Design methodology for fine-grained leakage control in MTCMOS," *International Symposium on Low Power Electronics and Design*, pp. 104-109, Aug. 2003.

[19] T. Kobayashi, and T. Sakurai, "Self-adjusting threshold-voltage scheme (SATS) for low-voltage high-speed operation," *IEEE Custom Integrated Circuits Conference*, pp. 271-274, May. 1994.

[20] M. Sumita, "High resolution body bias techniques for reducing the impacts of leakage current and parasitic bipolar," *International Symposium on Low Power Electronics and Design*, pp. 203-208, Aug. 2005.

[21] M. Horiguchi, T. Sakata, and K. Itoh, "Switched-source-impedance CMOS circuit for low standby subthreshold current giga-scale LSI's," *IEEE Journal of Solid-State Circuits*, vol. 28, pp. 1131-1135, Nov. 1993.

[22] T. Kawahara, M. Horiguchi, Y. Kawajiri, G. Kitsukawa, T. Kure, and M. Aoki, "Subthreshold current reduction for decoded-driver by self-reverse biasing," *IEEE Journal of Solid-State Circuits*, vol. 28, pp. 1136-1144, Nov. 1993.

[23] W.N. Li, A. Lim, P. Agrawal, and S. Sahni, "On the circuit implementation problem," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, pp 1147-1156, Aub. 1993.

[24] V. Sundararajan, and K.K. Parhi, "Low power synthesis of dual threshold voltage CMOS VLSI circuits," *International Symposium on Low Power Electronics and Design*, pp. 139-144, 1999.

[25] L. Wei, A. Chen, M. Johnson, K. Roy, and V. De, "Design and optimization of low voltage high performance dual threshold CMOS circuits," *ACM Design Automation Conference*, pp. 489-494, Jun. 1998.

[26] L. Wei, Z. Chen, K. Roy, M. Johnson, Y. Ye, and V. De, "Design and optimization of dual-threshold circuits for low-voltage low-power applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, pp. 16-24, Mar. 1999.

[27] Q. Wang, and S.B.K. Vrudhula, "Static power optimization of deep submicron CMOS circuits for dual V_T technology", *IEEE International Conference on Computer-Aided Design*, pp. 490-496, Nov. 1998.

[28] Y. Ho, and T. Hwang, "Low power design using dual threshold voltage," *ACM Design Automation Conference*, pp. 205-208, Jan. 2004.

[29] X. Tang, H. Zhou, and P. Banerjee, "Leakage power optimization with dual-Vth library in high-level synthesis," *Design Automation Conference*, pp. 202-207, Jun. 2005.

[30] K. Fujii, T. Douseki, and M. Harada, "A sub-1V triple-threshold CMOS/SIMOX circuit for active power reduction," *International Solid-State Circuits Conference*, pp. 190-191, Feb. 1998.

[31] K. Fujii, and T. Douseki, "A 0.5-V, 3-mW, 54x54-b multiplier with a triple-V_{th} CMOS/SIMOX circuit scheme," *IEEE International SOI Conference*, pp. 72-74, Oct. 1999.

[32] J. Bang-Jensen, G. Gutin, and A. Yeo, "When the greedy algorithm fails," *Descrete Optimization*, vol. 1, pp. 121-127, Nov, 2004.

[33] N. Tripathi, A. Bhosle, D. Samanta, and A. Pal, "Optimal assignment of high threshold voltage for synthesizing dual threshold CMOS circuits," *IEEE International Conference on VLSI Design*, pp. 227-232, Jan. 2001.

[34] Q. Wang, and S.B.K. Vrudhula, "Algorithms for minimizing standby power in deep submicrometer, dual-V_t CMOS circuits", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, pp. 306-318, Mar. 2002.

[35] K. Shin, and T. Kim, "Leakage power minimisation in arithmetic circuits," *Electronics Letters*, pp. 415-417, Apr. 2004.

[36] B. Chung, and J.B. Kuo, "Gate-Level Dual-Threshold Static Power Optimization Methodology (GDSPOM) for Designing High-Speed Low-Power SOC Applications Using 90nm MTCMOS Technology," *IEEE International Symposium on Circuits and Systems*, pp. 4, May 2006.

[37] B. Chung, *Gate-Level Dual-Threshold Static Power Optimization Methodology (GDSPOM) for Designing High-Speed Low-Power SOC Applications Using 90nm MTCMOS Technology*, M.A.Sc thesis, Simon Fraser University, Burnaby, BC, 2005.

[38] L.R. Ford Jr., and D.R. Fulkerson, *Flows in Networks*. Princeton, NJ: Princeton University Press, 1962.

[39] A.V. Goldberg, and R.E. Tarjan, "A new approach to the maximum flow problem," *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pp. 136-146, May 1986.

[40] P. Gupta, A.B. Kahng, and P. Sharma, "A practical transistor-level dual threshold voltage assignment methodology," *International Symposium on Quality of Electronic Design*, pp. 421-426, Mar. 2005.

[41] L. Wei, Z. Chen, K. Roy, Y. Ye, and V. De, "Mixed-V_{th} (MVT) CMOS circuit design methodology for low power applications," *ACM Design Automation Conference*, pp. 430-435, Jun. 1999.

[42] S. Sirichotiyakul, T. Edwards, C. Oh, J. Zuo, A. Dharchoudhury, R. Panda, and D. Blaauw, "Stand-by power minimization though simultaneous threshold voltage selection and circuit sizing," *ACM Design Automation Conference*, pp. 436-441, Jun. 1999.

[43] ----, *PrimeTime User Guide: Fundamentals*, Synopsys, Jun. 2005.

[44] H.I.A. Chen, E.K.W. Loo, J.B. Kuo, and M.J. Syrzycki, "Triple-threshold static power minimization technique in high-level synthesis for designing high-speed low-power SOC applications using 90nm MTCMOS technology," *Canadian Conference on Electrical and Computer Engineering*, Vancouver, BC, Apr. 2007.

[45] H.I.A. Chen, E.K.W. Loo, J.B. Kuo, and M.J. Syrzycki, "Triple-threshold static power minimization in high-level synthesis of VLSI CMOS," *International Workshop on Power and Timing Modeling, Optimization and Simulation*, Goteborg, Sweden, pp. 453-462, Sep. 2007.

[46] O. MacSorley, "High speed arithmetic in binary computers," *IRE Proceedings*, vol. 49, pp. 67-91, Jan. 1961.

[47] C. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on Electronic Computers*, EC-13, pp. 14-17, Feb. 1964.

[48] P.M. Kogge, and H.S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Transactions on Computers*, vol. C-22, pp. 786-793, Aug. 1973.

[49] F. Brglez, and H. Fujiwara, "A neutral netlist of 10 combinational benchmark circuits and a target translator in Fortran," *IEEE International Symposium on Circuits and Systems*, Jun. 1985.

[50] F. Brglez, D. Bryan, and K. Kozminski, "Combinatorial profiles of sequential benchmark circuits," *IEEE International Symposium on Circuits and Systems*, pp. 1929-1934, 1989.

[51] F. Corno, M.S. Reorda, and G. Squillero, "RT-level ITC'99 benchmarks and first ATPG results," *IEEE Design & Test of Computers*, vol. 17, pp. 44-53, Jul. 2000.