Power-Efficient Design of 16-Bit Mixed-Operand Multipliers

by

Sataporn Pornpromlikit

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2004
[June 2004]

Author..
Sataporn Pornpromlikit
Department of Electrical Engineering and Computer Science
May 20, 2004

Certified by..
Victor Zyuban
VI-A Company Thesis Supervisor

Certified by..
Thomas F. Knight Jr.
M.I.T. Thesis Supervisor

Accepted by..
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Power-Efficient Design of 16-Bit Mixed-Operand Multipliers

by

Sataporn Pornpromlikit

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Multiplication is an expensive and slow arithmetic operation, which plays an important role in many DSP algorithms. It usually lies in the critical-delay paths, having an effect on performance of the system as well as consuming large power. Consequently, significant improvements in both power and performance can be achieved in the overall DSP system by carefully designing and optimizing power and performance of the multiplier. This thesis explores several circuit-level techniques for power-efficiently designing multipliers, including supply voltage reduction, efficient multiplication algorithms, low power circuit logic styles, and transistor sizing using dynamic and static tuners. Based on these techniques, several 16-bit multipliers have been successfully designed and implemented in 0.13μm CMOS technology at the supply voltage of 1.5V and 0.9V. The multipliers are modified to handle multiplications of two 16-bit operands in which each can be either signed magnitude or two's complement formats. Examining power-performance characteristics of these multipliers reveals that both array and tree structures are feasible solutions for designing 16-bit multipliers, and complementary CMOS and single-ended CPL-TG logics are promising candidates for power-efficient design. The appropriate choices of structures and logic styles depend on power and performance constraints of the particular design.

VI-A Company Thesis Supervisor: Victor Zyuban
Title: Research Staff Member, IBM T.J. Watson Research Center

M.I.T. Thesis Supervisor: Thomas F. Knight Jr.
Title: Senior Research Scientist

Acknowledgements

First of all, I would like to thank my mentor and thesis supervisor Dr. Victor Zyuban for his wonderful guidance throughout my time at IBM. He is the one who started all the work and made things so easy for me. I am amazed by his intelligence and work ethics, and knowing him helps me improve as a researcher.

I would like to thank Dr. Tom Knight for being a patient and generous MIT faculty supervisor. My knowledge and interest in the field began from taking his class, and I am forever honored to be one of his students. Apart from his excellence in teaching and research, he has always been very kind and easy to get along with.

Special thanks to all IBM research staff, especially my manager Dr. Jaime Moreno for giving me the opportunity to work in the group, Tom Fox for his guidance and support in my first year with the group, Jude Rivers for his friendliness, Sameh Asaad for his help on VHDL verification and several other things, Ann-Marie Haen for her EinsTuner support, Kelvin Lewis for the ModelSim setup and license issue, and George Gristede for his support on SST circuit tuning tool. Without these people, my work would never be materialized.

I also want to mention several people whose friendships, encouragements, and supports help me survive MIT: Elcid, Bo, Note, Pop, Pui, Poo'Ay, Pann, Hok, Cee, Kua, Sup, Ton, Vee, Wang, Petch, Bua, Boss, Pep, Zhenye, Cang-Kim Truong, Yao, Hiro, Mamat, Esosa, and all other NH4 and TSMIT friends. I will always remember the good times we have shared.

To my family, Akong, Ama, Papa, Mama, E'Yai, E'MuayLek, E'Eng, E'Ngor, E'Muay, E'Dang, Gu+Chiang, Gu+Chai, Gu+Yai, Gu+Dum, Gu+Meng, Gu+Lek, Jin, An, Ek, Oil, Ice, Em, Jay, May, Pun, Yoyo, Pingpong, Mint, AngPao, and Pin: You are the main reason I live my life today, and I am enjoying every minute of it.

Contents

Chapter 1

Introduction

1.1 Motivation

Minimizing the power consumption has become of great important in modern integrated circuit design due to the emergence of high-performance portable electronics and personal communication systems. A number of such applications have been developed everyday with increasing demand for power reduction to extend the battery life and reduce its weight. Consequently, designing circuits and systems for low power has been an important research theme in recent years.

There have been several research efforts in literature focusing on designing low-power high-performance digital signal processor (DSP), which is the core of today's high-performance communication systems. The motivation of this project stems from one such effort at IBM to explore methodologies for power-efficient DSP design at all levels of abstraction, ranging from algorithms, applications, and high-level language compiler, down to circuit-level technology.

Specifically, the project aims to explore methodologies for power-efficient design at circuit level by using the 16-bit multiplier as a vehicle. Multiplication is an expensive and slow arithmetic operation, which plays an important role in many DSP algorithms. It usually lies in the critical-delay paths, having an effect on performance of the system as well as consuming large power. Significant improvements in both power and performance can be achieved in the overall system by carefully optimizing power and performance of the multiplier.

Several techniques for designing power-efficient multipliers to be explored include transistor sizing using dynamic and static tuners, a proper selection of circuit logic styles, and efficient multiplication algorithms. Unlike most existing multipliers, the 16-bit multipliers designed in this project are modified to handle multiplications of two 16-bit operands in which each can be either signed magnitude or two's complement formats. This modification allows the multipliers to be used for multiplying an unsigned 32-bit number by a signed 16-bit number, which is required in many DSP algorithms [1].

1.2 Thesis Organization

Chapter 1 gives a brief overview of the motivation and goal of this thesis project. Chapter 2 discusses the mechanisms for power dissipation in CMOS circuits, which are crucial for understanding the intuitions behind the approaches and techniques for power reduction employed in this thesis. Chapter 3 looks into some static logic styles that are potentially suitable for low-power design, such as complementary CMOS logic, pass-transistor logic, and transmission-gate logic. Based on these logic styles, several full-adder circuit implementations, which are the basis for almost every arithmetic unit including the multiplier, are presented. Chapter 4 explains the basic concept of multiplication, as well as briefly reviews existing multiplication algorithms found in literature. We also presents an efficient algorithm for modifying a multiplier to handle multiplication of two operands in which each can be either unsigned magnitude or two's complement formats. Chapter 5 describes the design of 16-bit multipliers, both array and tree configurations, using the signed-unsigned algorithm proposed in Chapter 4 and several full-adder designs implemented in Chapter 3. The implemented 16-bit multipliers are, then, optimized using both dynamic and static tuning techniques explored in Chapter 6. Results are presented and discussed in Chapter 7, where the impacts of multiplication structures and circuit logic styles on the power-performance characteristics of 16-bit multipliers are investigated. The thesis concludes in Chapter 8, which summarizes the important points obtained from this project.

Chapter 2

Power Dissipation in CMOS Circuits

This chapter discusses the mechanisms for power dissipation in CMOS circuits, as well as provides general intuitions and design techniques for power reduction.

2.1 Sources of Power Dissipation in CMOS Circuits

The power dissipation in CMOS circuits can be described by:

$$P_{avg} = P_{dynamic} + P_{short-circuit} + P_{static} \tag{2.1}$$

where P_{avg} is the average power dissipation, $P_{dynamic}$ is the dynamic power dissipation due to the switching of transistors, $P_{short\text{-}circuit}$ is the power dissipation due to the short-circuit current, and P_{static} is the static power dissipation.

2.1.1 Dynamic Power Dissipation

The dynamic power dissipation is caused by the charging and discharging of capacitances in the circuit. To illustrate the process, consider the CMOS inverter in Figure 2.1, where the output capacitor C_L is due to parasitic capacitances of the NMOS and PMOS transistors, wire capacitance, and the input capacitance of the circuits driven by the inverter.

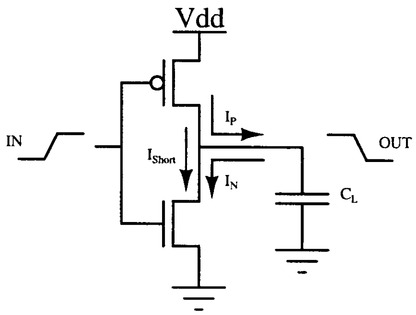


Figure 2.1: Power dissipation in CMOS Inverter.

As the input switches from V_{dd} to ground, the NMOS is cut off and the PMOS is switched on charging C_L up to V_{dd}. Then, when the input returns to V_{dd}, the process is reversed and C_L is discharged. This one cycle of charging and discharging C_L draws energy equal to $C_LV_{dd}^2$ from the power supply. The average dynamic power dissipation can be described by:

$$P_{dynamic} = \alpha C_L V_{dd}^2 f \tag{2.2}$$

where α is the switching activity with the value ranging from zero to one transition per data cycle, and *f* is the average data rate or the clock frequency in a synchronous system.

The dynamic power dissipation is the dominant factor compared with the other components of power dissipation in digital CMOS circuits. As technology scales down for submicron technologies, the contribution of dynamic power dissipation also increases due to increased functionality requirements and the clock frequencies [7]. Consequently, the majority of existing low power design and power reduction techniques focuses on this dynamic component of dissipation. For instance, lower supply voltages are becoming more and more attractive since it has a quadratic effect on dynamic power dissipation.

2.1.2 Short-Circuit Power Dissipation

The short-circuit power dissipation is caused by the current flowing through the direct path between the power supply and the ground during the transition phase. Consider again the CMOS inverter shown in Figure 2.1. When the input voltage is between V_{tn} and V_{dd}-$|V_{tp}|$ during the switching, both NMOS and PMOS devices will be turned on, allowing short-circuit current to flow. The short-circuit power dissipation of a CMOS inverter can be approximated by [8]:

$$P_{short-circuit} = K \cdot (V_{dd} - 2V_{th})^3 \cdot \tau \cdot N \cdot f \tag{2.3}$$

where K is the constant that depends on the transistor sizes, as well as on the technology, V_{th} is the threshold voltage of the NMOS and PMOS devices, τ is the rise and fall time of the input signal, N is the average number of transitions in the inverter's output, and *f* is the average data rate.

The short-circuit power dissipation is minimized by matching the rise and fall times of the input and output signals. At the overall circuit level, the short-circuit power dissipation can be kept below 10% of the total power by keeping the rise and fall times of all the signals throughout the design within a fixed equal range. The impact of short-circuit current is also reduced as the supply voltage is decreased. In the extreme case where $V_{dd}<V_{thn}+|V_{thp}|$, short-circuit power dissipation is completely eliminated because both devices are never on simultaneously. With threshold voltages scaling down at a slower rate than the supply voltage, short-circuit power dissipation is becoming less important [6].

Note that, in dynamic logic gates, there is no direct current path between the power supply and the ground nodes because the pre-charged and evaluated transistors should never be on simultaneously in order to function correctly [8].

2.1.3 Static Power Dissipation

Ideally, CMOS circuits dissipate no static power since in the steady state there is no direct path from V_{dd} to ground. Unfortunately, MOS transistor is not a perfect switch, and in reality, power could be dissipated even when the transistors are not performing any switching due to the following factors: leakage currents, degraded input signals of the complementary logic gates, and static power dissipation in ratioed logic families.

Leakage Currents

The leakage power dissipation is caused by two types of leakage currents: the reverse-bias diode leakage current at the transistor drains and the sub-threshold current through turned-off transistor channel [8]. Diode leakage current occurs when a transistor is turned off and its drain is charged up/down by the other ON transistor such that the diode formed by the drain diffusion and the substrate is reversed-bias. The sub-threshold leakage current occurs due to carrier diffusion between the source and the drain of MOS transistor operating in the weak inversion region (below the threshold voltage).

Degraded Input Signals

As illustrated by Figure 2.2, the voltage level at node A is degraded by the voltage threshold of the pass transistor. When the input signal is high at V_{dd}, the inverter's input is $V_{dd}-V_{thn}$, and the PMOS transistor is not completely off causing a static current from the power supply to the ground.

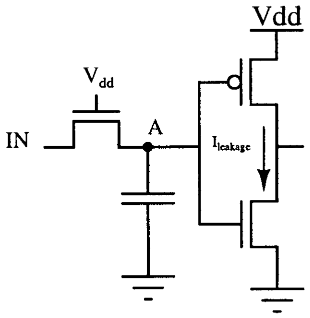


Figure 2.2: Static power dissipation due to degraded input signals.

Ratioed Logic Families

Another form of static power dissipation occurs when ratioed logic families are used [6]. A pseudo-NMOS logic gate, for example, consists of a complex block of NMOS transistors implementing the Boolean function and a single always-ON PMOS transistor (Figure 2.3). Clearly, there always exists a direct path from the power supply to the ground causing static power dissipation during the steady state. This type of static power dissipation could be considerable, and thus, the ratioed logic families should be avoided when designing low-power CMOS circuits.

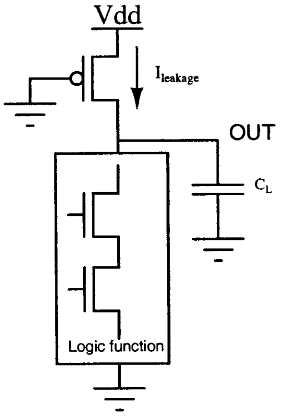


Figure 2.3: Static power dissipation in a pseudo-NMOS logic gate.

In summary, the static component of power dissipation should be negligible in low-power CMOS circuits, and the short-circuit power dissipation can be kept within less than 10% by a careful design. The dynamic power dissipation is by far the dominant factor in typical CMOS circuits, and thus, the main focus in realizing power reduction techniques.

2.2 Dynamic Power Reduction Approaches

As described in section 2.1.1, the average dynamic power dissipation of CMOS circuits is proportional to the square of the power supply V_{dd}, the physical capacitance C_L, the switching activity α, and the average data rate f. Thus, the fundamental concept for dynamic power reduction involves lowering a combination of the variables above. The reduction of power supply voltage is one of the most aggressive techniques due to its quadratic relation. It is often reasonable to increase physical capacitance and switching activity in order to further reduce supply voltage. Unfortunately, the supply voltage can only be decreased to some extent due to performance requirements and compatibility issues.

The load capacitance can be reduced by transistor sizing, selection of the proper logic styles, placement and routing, and architectural optimization. With a detailed analysis of signal transition probabilities, the switching activity can be reduced at all levels of the design abstraction including logic restructuring, input ordering, glitch reduction by balancing signal paths [6].

Chapter 3

Circuit Techniques and Logic Styles for Full Adder Design

As described in the previous chapter, the average power dissipation of the circuits is mainly determined by the switching activities, the load capacitances, and the short-circuit currents. The circuit speed is determined by characteristics such as transistor sizes, the number of transistors in series, and the wiring complexity. Because these characteristics can vary considerably from one logic style to another, selection of proper logic styles can potentially improve the overall performance and average power dissipation of the circuits.

This chapter briefly discusses some static logic styles that are potentially suitable for low-power design, such as complementary CMOS logic, pass-transistor logic, and transmission-gate logic. We then implement several full-adder circuits, which are the basis for almost every arithmetic unit including the multiplier, based on these logic styles. Dynamic logic style is attractive for high-speed application; however, it may not be a good candidate for low-power applications due to its large clock loads and high switching activities caused by the pre-charging mechanism, and will not be further investigated in this thesis.

3.1 The Full Adder

The full-adder circuit is the logic circuit that takes three binary inputs, A, B, and C_i (*Carry-in* bit), and provides two binary outputs, S (*Sum* bit) and C_o (*Carry-out* bit). This circuit is sometimes called a 3-2 compressor. The Boolean expressions for S and C_o are given as:

$$\begin{aligned} S &= A \oplus B \oplus Ci \\ &= A\overline{B}\,\overline{Ci} + \overline{A}B\overline{Ci} + \overline{A}\,\overline{B}Ci + ABCi \\ Co &= AB + BCi + ACi \end{aligned} \tag{3.1}$$

S and C_o can also be defined as functions of intermediate signals G (*Generate*) and P (*Propagate*) as the followings:

$$G = AB$$

$$P = A \oplus B$$

$$S(G,P) = P \oplus Ci$$

$$Co(G,P) = G + PCi \tag{3.2}$$

3.2 Complementary CMOS Logic

The static complementary CMOS logic is a combination of two networks, the pull-up network and the pull-down network. The pull-up network, composed of only PMOS devices, provides a connection between the output and the V_{dd} whenever the output is supposed to be high ('1'), while the pull-down network, composed of only NMOS devices, provides a connection between the output and the ground node whenever the output is supposed to be low ('0'). The pull-up network and the pull-down network are dual and mutually exclusive such that only one network is conducting at the steady state.

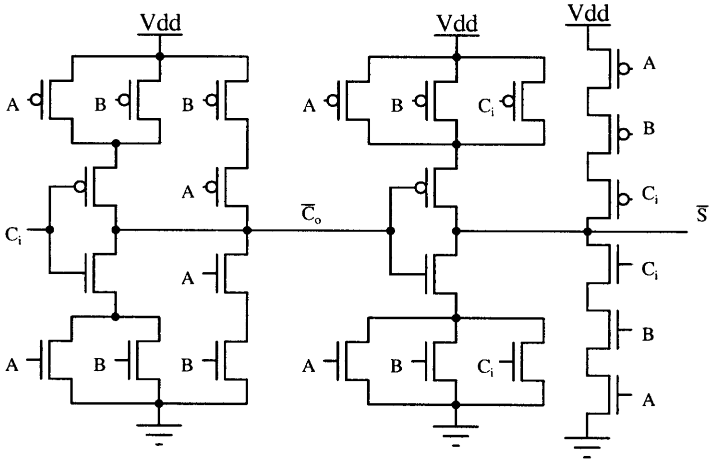


Figure 3.1: Complementary CMOS full adder: mirror adder schematic.

The full-adder circuit can be implemented in complementary CMOS logic by directly translating the Boolean expressions for S and C_o given in Eq. (3.1). However, the improved adder circuit, called *mirror adder*, can be obtained using the S(G,P) and C_0(G,P) functions described in Eq. (3.2), as shown in Figure 3.1. The circuit requires 24 transistors, and a maximum of only two transistors in series can be found in the carry-generation circuitry [6].

3.3 Pass-Transistor Logic

Pass-transistor logic has been widely used as an alternative to complementary CMOS logic in low-power applications. It allows the inputs to drive gate terminals as well as source-drain terminals, requiring less numbers of transistors to implement logic functions. Figure 3.2 shows some examples of pass-transistor networks, consisting of only NMOS transistors. The networks are constructed in a tree-like form, consisting of a sequence of 2-way multiplexers.

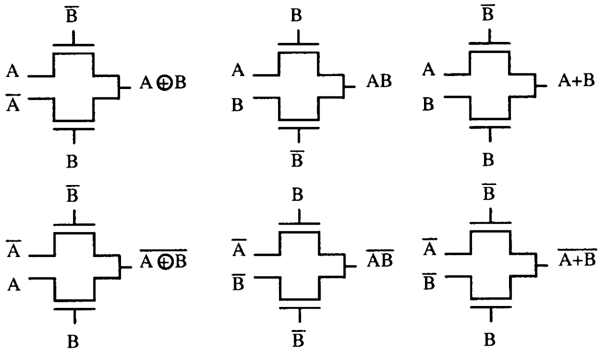


Figure 3.2: Examples of pass-transistor networks.

3.3.1 Single-Ended Pass-Transistor Logic (SPL or LEAP)

SPL logic, also know as LEAP (Lean Integration with Pass-Transistors), is the simplest of the pass-transistor logic family. It was originally proposed to establish an automated design methodology for forming logic functions using pass-transistors [9]. The logic is single-ended, and complementary signals can be generated locally by inverting selected nodes.

Figure 3.3 shows a full adder implemented using SPL logic style. Since the high input to the signal-restoring inverter is degraded (only charged up to V_{dd}-V_{thn}), there is a leakage current flowing through the inverter causing static power dissipation as explained in Chapter 2. One way to solve this problem is to add a *level-restorer*, which is a weak PMOS transistor configured in a feedback path across the input and the output of the inverter. When the input to the inverter is charged up to V_{dd}-V_{thn}, which is enough to switch the output of the inverter low, the PMOS device is turned on pulling the input node of the inverter all the way up to V_{dd}.

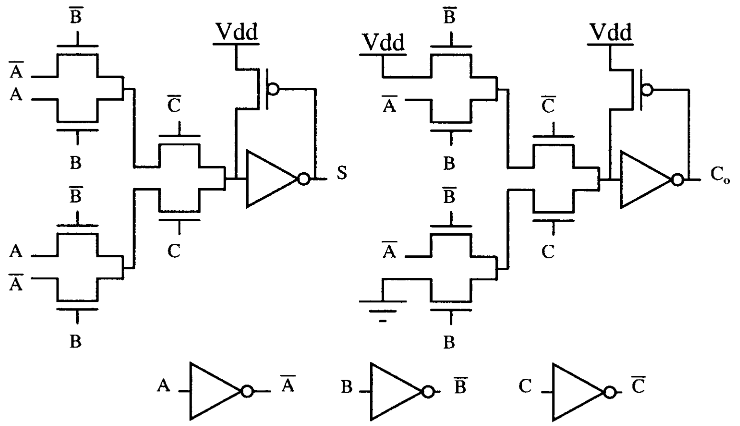


Figure 3.3: SPL full adder.

3.3.2 Complementary Pass-Transistor Logic (CPL)

CPL has been the most widely used in high-performance low-power applications among the pass-transistor logic configurations. Unlike the SPL circuits, CPL circuits are differential, and thus, complementary inputs and outputs are always available, eliminating the need for some extra inverters. It also prevents some time-differential problems causing by additional inverters. However, one drawback is that the number of wires to be routed is virtually doubled. The dynamic power dissipation is also potentially higher than the single-ended [6].

Figure 3.4 illustrates a full-adder circuit implemented using CPL logic style. Similar to SPL circuit, the level-restorer is needed in each network to prevent static power dissipation due to the leakage current through the inverter. Since the output signals are complementary, the cross-coupled PMOS devices are used as the level-restorer. These PMOS devices, however, reduce performance slightly by adding extra evaluate capacitance [5].

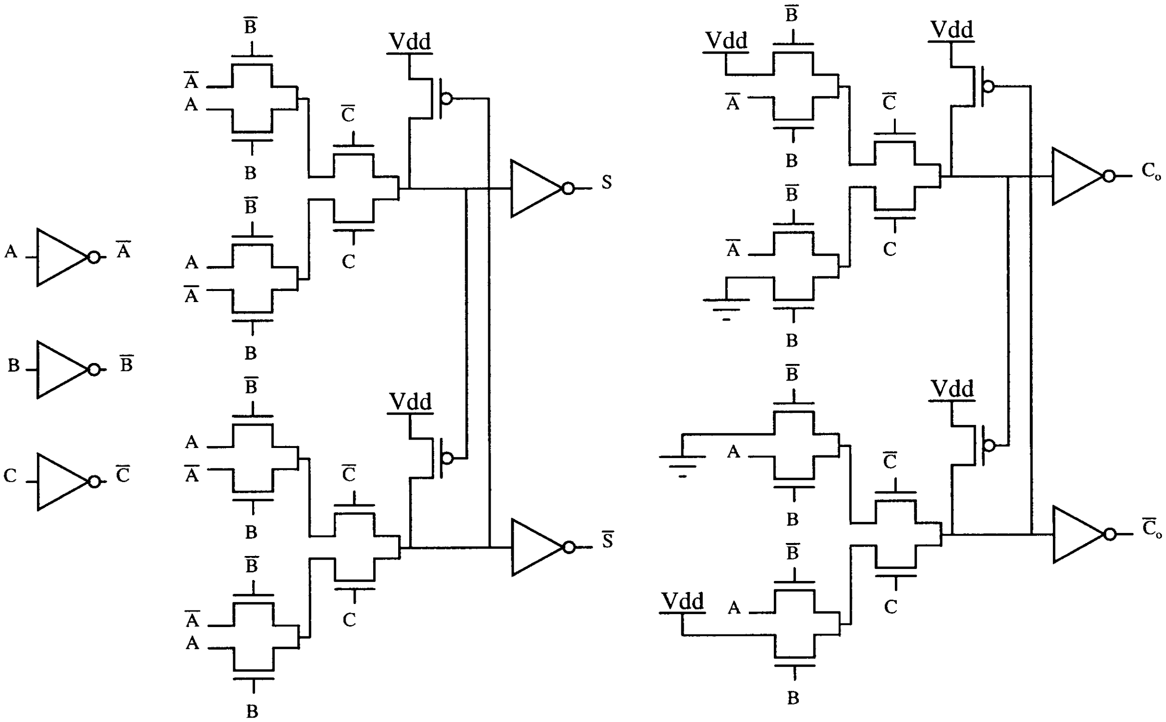
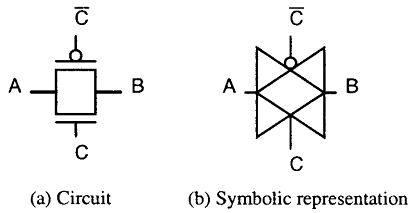


Figure 3.4: CPL full adder.

3.4 Transmission-Gate Logic

Another widely-used method to solve the signal degradation problem due to pass-transistors is the use of transmission gates. This technique combines the complementary properties of the NMOS and PMOS devices (NMOS devices pass a strong ‘0’, but a weak ‘1’, while PMOS devices pass a strong ‘1’, but a weak ‘0’) by placing an NMOS device and a PMOS device in parallel [6] as shown in Figure 3.5. The transmission gate is a bidirectional switch controlled by the gate signal C, where A=B if C=1. Suppose the transmission gate is enabled (C=1) and both NMOS and PMOS transistors are on. If node A is set at V_{dd}, the output node charges all the way up to V_{dd} through the PMOS transistor. If node A is set low, the output node discharges all the way down to ground through the NMOS transistor. Without the PMOS, the output node will only charge up to V_{dd}-V_{thn}. Similarly, the output node will only discharge down to V_{thp} without the NMOS transistor.



(a) Circuit (b) Symbolic representation

Figure 3.5: CMOS transmission gate.

With the use of transmission-gate logic, the CPL full-adder circuit in Figure 3.4 can be modified as shown in Figure 3.6 (Single-ended CPL-TG) and Figure 3.7 (Dual-rail CPL-TG), based on the Propagate-Generate Boolean functions in Eq. (3.2). The *propagate* signals are generated using cross-coupled CPL logic. Then, the transmission gates are used to produce the *sum* and the *carry* output signals. The single-ended CPL-TG full adder in Figure 3.6 has single-ended outputs. The dual-rail CPL-TG full adder in Figure 3.7 has single-ended *sum* output and complementary *carry* outputs.

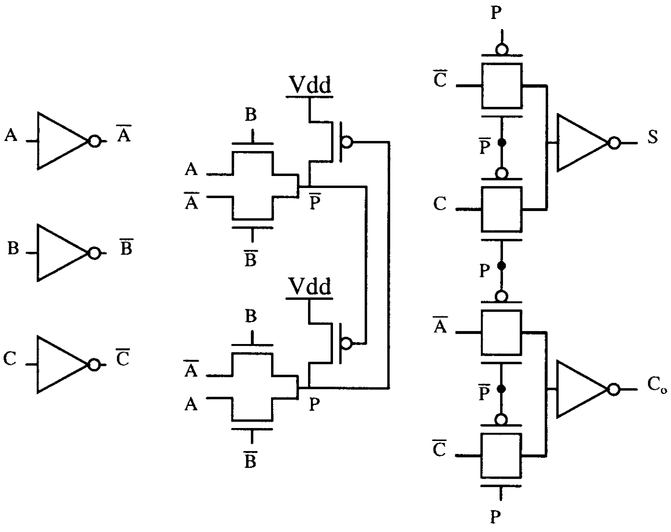


Figure 3.6: Single-ended CPL-TG full adder.

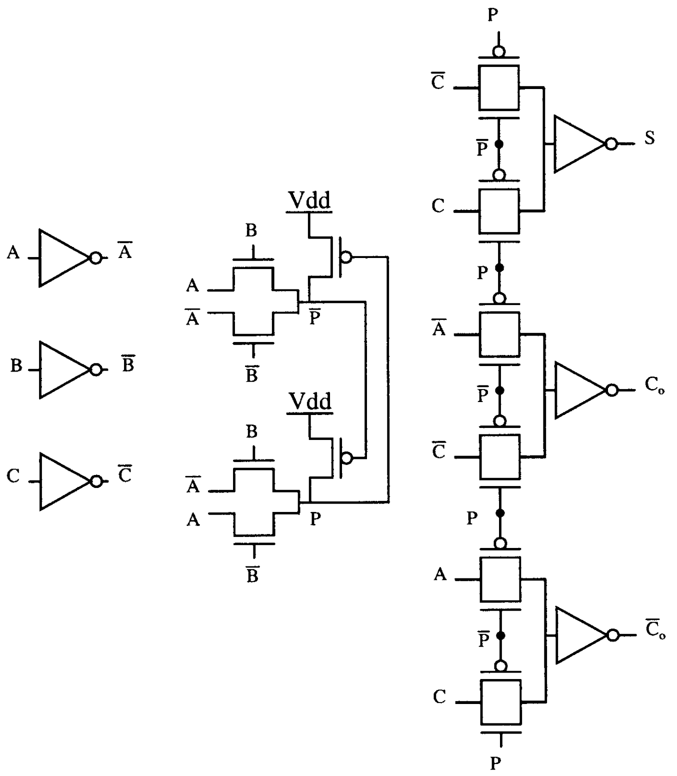


Figure 3.7: Dual-rail CPL-TG full adder.

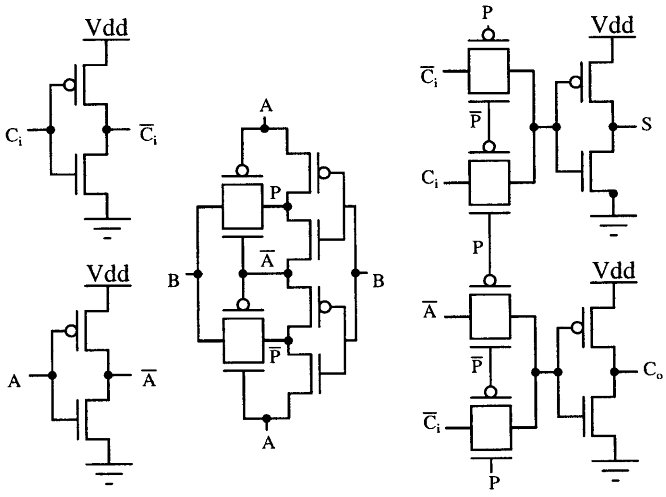


Figure 3.8: Transmission-gate-based full adder.

The transmission gates can also be used to efficiently build some complex gates such as 2-input multiplexers and XOR gates. Consequently, the full adder can be implemented using only transmission-gate-based multiplexers and XOR gates as shown in Figure 3.8.

Chapter 4

The Multiplier

As explained later in this chapter, multipliers can basically be viewed as complex adder arrays operating in three stages: partial-product generation, partial-product accumulation, and final adder. There are several multiplier algorithms found in literature, but they can be classified into two main categories based on the structure of their partial-product accumulation: array multiplier and tree multiplier. This chapter discusses these topics, as well as presents an efficient algorithm for modifying a multiplier to handle multiplication of two operands of which each can be in either unsigned magnitude or two's complement formats.

4.1 Basic Concept

Consider the following two unsigned binary numbers X and Y, where $X_i, Y_j \in \{0,1\}$.

$$X = \sum_{i=0}^{m-1} X_i 2^i \quad Y = \sum_{j=0}^{n-1} Y_j 2^j \tag{4.1}$$

The multiplication operation is defined as:

$$P = X \times Y = \sum_{i=0}^{m-1} [\sum_{j=0}^{n-1} X_i Y_j 2^{i+j}] \tag{4.2}$$

One effective way to implement a multiplier is to simulate the manual operation illustrated in Figure 4.1. All partial products can be generated simultaneously and organized in an array. Then, the partial products are systematically accumulated to produce the final result. Based on this approach, the multiplier may be viewed to consist of three main components: partial-product generation, partial-product accumulation, and final addition.

Generation of partial products consists of the logical AND operations of the relevant operand (multiplicand and multiplier) bits. Each column of partial products must then be compressed with any carry bit passed to the next column. The final step is to combine the result in the final adder.

$$
\begin{array}{cccccccccc l}
 & & & & 1 & 0 & 1 & 1 & 0 & 1 & \text{Multiplicand} \\
\text{x} & & & & & & 1 & 0 & 1 & 1 & \text{Multiplier} \\
\hline
\end{array}
$$

$$
\left.\begin{array}{cccccccccc}
 & & & & 1 & 0 & 1 & 1 & 0 & 1 \\
 & & & 1 & 0 & 1 & 1 & 0 & 1 & \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & & \\
+ & 1 & 0 & 1 & 1 & 0 & 1 & & & \\
\hline
\end{array}\right\} \text{Partial products}
$$

$$
\begin{array}{cccccccccc l}
 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & \text{Result}
\end{array}
$$

Figure 4.1: Binary multiplication.

4.2 Prior-Art Multipliers

Based on how the partial products are accumulated, most existing multipliers can be classified into two main categories: an array multiplier and a tree multiplier. The following sections present how these multipliers are implemented, and briefly discuss some techniques, such as Baugh-Wooley algorithm and Booth's recoding algorithm, that are widely used to modify the multiplier.

4.2.1 Array Multiplier

An array multiplier can be implemented by directly mapping the manual multiplication into hardware as explained in the previous section. The partial products are accumulated by an array of adder circuits as shown in Figure 4.2, thus, the name *array multiplier* has been adopted.

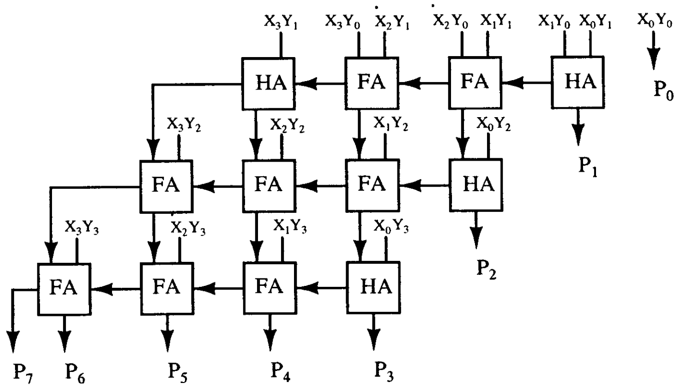


Figure 4.2: A 4x4-bit array multiplier. HA represents an adder with only two inputs, or a half adder.

An nxn array multiplier requires n(n-1) adders and n^2 AND gates. The delay of the multiplier is dependent on the delay of the full-adder cell and the final adder in the last row. A full-adder with balanced carry-out and sum delays is desirable because both signals are on the critical path.

Baugh-Wooley Algorithm

Note that the array multiplier previously discussed only performs multiplication of two unsigned numbers. For multiplication of two numbers in two's complement format, the Baugh-Wooley algorithm was commonly used to modify the unsigned multiplier [4]. However, the Baugh-Wooley scheme becomes slow and area-consuming for multipliers with operands equal to or greater than 16 bits.

Modified Booth's Recoding

For operands equal to or greater than 16 bits, Modified Booth's recoding has been widely used to reduce the number of partial products to be added. The algorithm is equivalent to transforming the multiplier from the binary format into a base-4 format. The scheme is modified from the original Booth's recoding to avoid a variable-size partial-product array [6].

4.2.2 Wallace Tree Multiplier

For large operands (32 bits and over), a Wallace tree multiplier can offer substantial hardware savings and faster performance [4]. Unlike an array multiplier, the partial-product matrix for a tree multiplier is rearranged and accumulated in a tree-like format, reducing both the critical path and the number of adder cells needed. It can be shown that the propagation delay through the tree is equal to $O(\log_{(3/2)}N)$ using 3-2 compressors (full adders).

There are numerous ways to implement a Wallace tree multiplier. The main design challenge is to realize the final result with a minimum depth and a minimum number of adder elements. Consider an example illustrated in Figure 4.3. A full adder (FA) is represented by a box covering three bits, and a half adder (HA) is represented by a box covering two bits. At each stage, an adder (either FA or HA) takes its covered input bits and produces two output bits: the *sum*, placed in the same column, and the *carry*, sent to the next column. At the first stage, two HAs are used in column 3 and 4. The reduced tree is further compressed in the second stage using three FAs and one HAs. The final addition is then performed at the last stage to produce the final result.

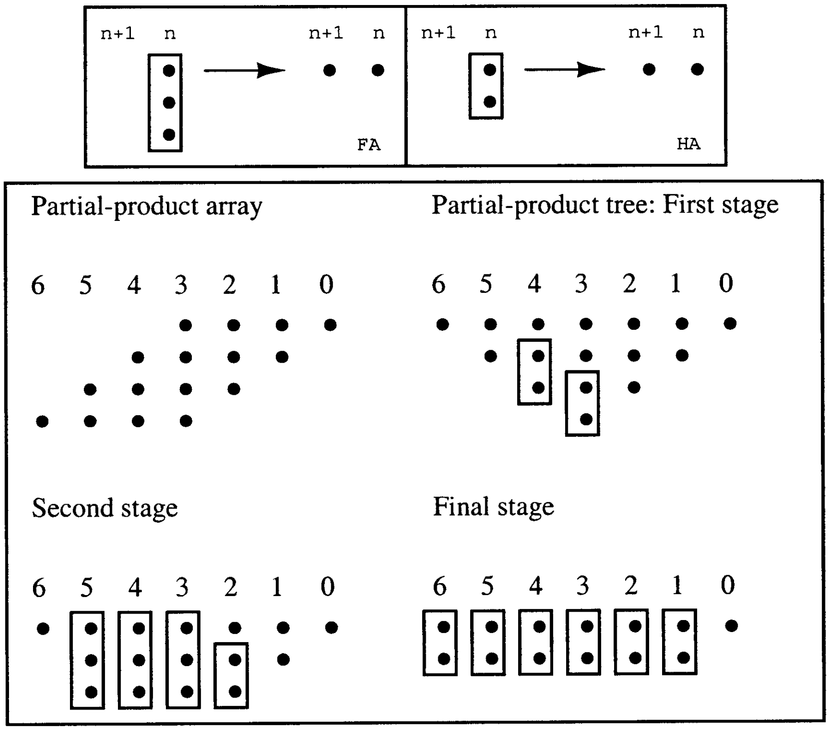


Figure 4.3: An example of 4x4 tree multiplier process.

4.2.3 Array Multiplier V.S. Tree Multiplier

The array multiplier generally exhibits low power dissipation and relatively good performance. The structure is regular and each cell is connected only to its neighboring cells, resulting in a compact layout and a simple wiring scheme. The modified Booth's recoding can be used to reduce the number of partial products for operands of 16 bits and over. For operands of larger word lengths, the Wallace tree multiplier realizes faster speed and considerable hardware savings. The main drawback of the tree structure, however, is its irregular layout and complex wiring scheme. In fact, several multipliers have been designed and compared in literature. However, the appropriate choice of structure, especially for a 16-bit multiplier, is not straightforward, depending on the power and the performance constraints of the design.

4.3 Signed-Unsigned Multiplication

The existing multipliers are either unsigned multipliers that accept two unsigned operands, or signed multipliers that accept two signed operand. A common approach used to multiply mixed-format operands, where the multiplier and the multiplicand can be either signed or unsigned, is to extend both operands to n+1 bits and use a signed n+1 by n+1 multiplier. Any unsigned n-bit number can be represented as a signed 2's complement (n+1)-bit number by adding zero as the most significant bit. Any signed n-bit number can be represented as a signed (n+1)-bit number by sign-extending it by one bit. Figure 4.4 illustrates an example of such approach.

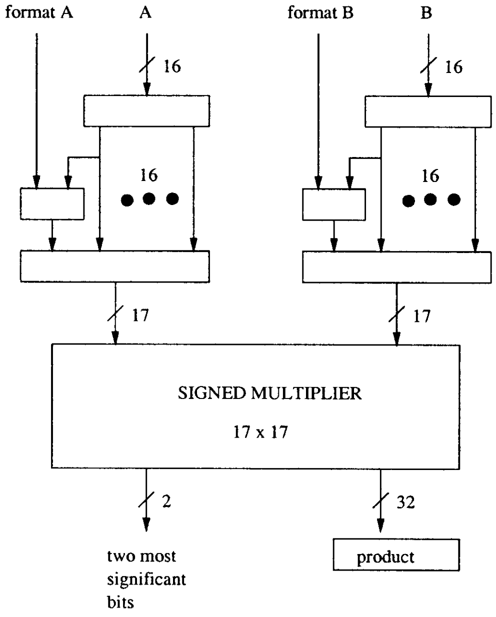


Figure 4.4: Using 17x17-bit signed multiplier to multiply 16-bit mixed-format operands.

However, the use of a larger (n+1)-bit signed multiplier could increase the power dissipation of the multiplier and could also increase the critical-path delay through the multiplier. An alternative approach used in this thesis is to modify a standard n-bit unsigned multiplier based on the algorithm proposed by J.H. Moreno et al. [1].

The Modified Algorithm for Signed-Unsigned Multiplication

Consider Figure 4.5. The shifted partial-product matrix is divided into four sections *(1)-(4)*. Similar to the standard unsigned multiplication, all partial products are normally computed using AND operation. The major difference is the inversion of the partial products in section *(2)*, *(3)*, and *(4)* controlled by the formats of operand A and B. Section *(2)* represents partial products of the form A_iB_n, where i is an index from 0 to n-1 and B_n is the most significant bit of operand B. The partial products in section *(2)* must be inverted if operand B is in signed format. Section *(3)* represents partial products of the form A_nB_j, where j is an index from 0 to n-1 and A_n is the most significant bit of operand A. The partial products in section *(3)* must be inverted if operand A is in signed format. Section *(4)* represents the product of the most significant bits of both operands, A_nB_n. The product in section *(4)* must be inverted if the two operands are in different formats.

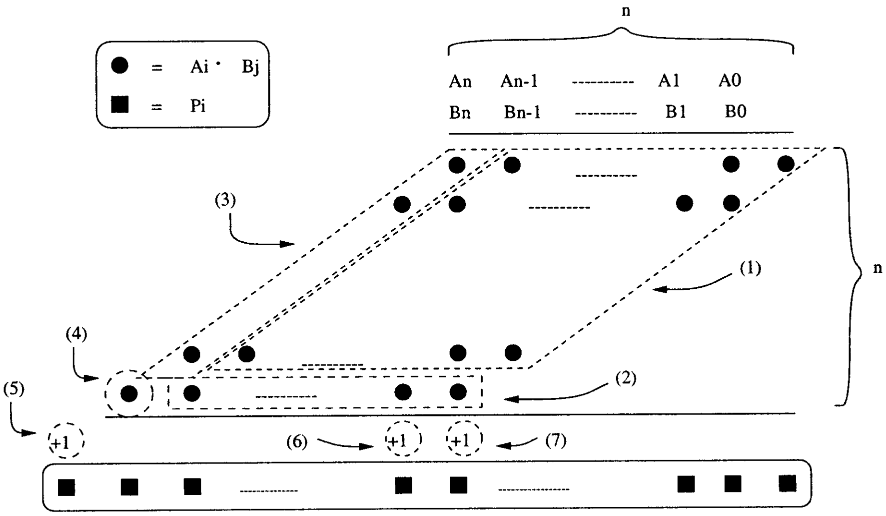


Figure 4.5: n-bit signed-unsigned multiplication: array-format partial-product matrix.

In addition to the inversion of partial products as described above, if only one of the operands is in signed format, an extra '1' is added to the final product at the n^{th} position from the right. If both operands are in signed format, an extra '1' is added to the final product at the $(n+1)^{th}$ position from the right. Finally, if one or both of the operands is in signed format, an extra '1' is added to the most significant bit of the final product.

Chapter 5

16-Bit Multiplier Design and Simulation

The appropriate choice of structures, array or tree, depends on the power and the performance constraints of the design. For each structure, the design solution can be optimized with appropriate circuit logic styles and transistor sizing. We will design and investigate 16-bit multipliers using both array and tree configurations, based on the signed-unsigned algorithm proposed in Chapter 4. The circuits are designed in 0.13μm CMOS technology at the supply voltages of 0.9V and 1.5V. Circuit logic styles are also compared in the context of a multiplier, which is a relatively complex arithmetic unit with thousands of possible critical paths.

5.1 Array Multiplier Design

An array multiplier is designed and modified based on the signed-unsigned algorithm illustrated in Figure 4.5. The operands can be in either unsigned magnitude or two's complement format. The circuit block diagram, shown in Figure 5.1, is very regular and easily fitted into a 16-bit datapath. All partial product terms are generated in parallel with AND gates that are mostly placed inside multiplier cells. Multiplication is done in two stages, where the second stage is a 16-bit adder.

Multiplier Cell

Apart from the AND gates on the first row of the circuit, each multiplier cell consists of an AND gate and either a full adder (FA) or a half adder (HA). The AND gate generates a partial product bit, and then, the FA adds this bit to the sum and carry bits (or only the sum bit for HA) from the previous multiplier cell. A multiplexer to selectively invert the partial product bit is included in the multiplier cells in the last row and the left-most column.

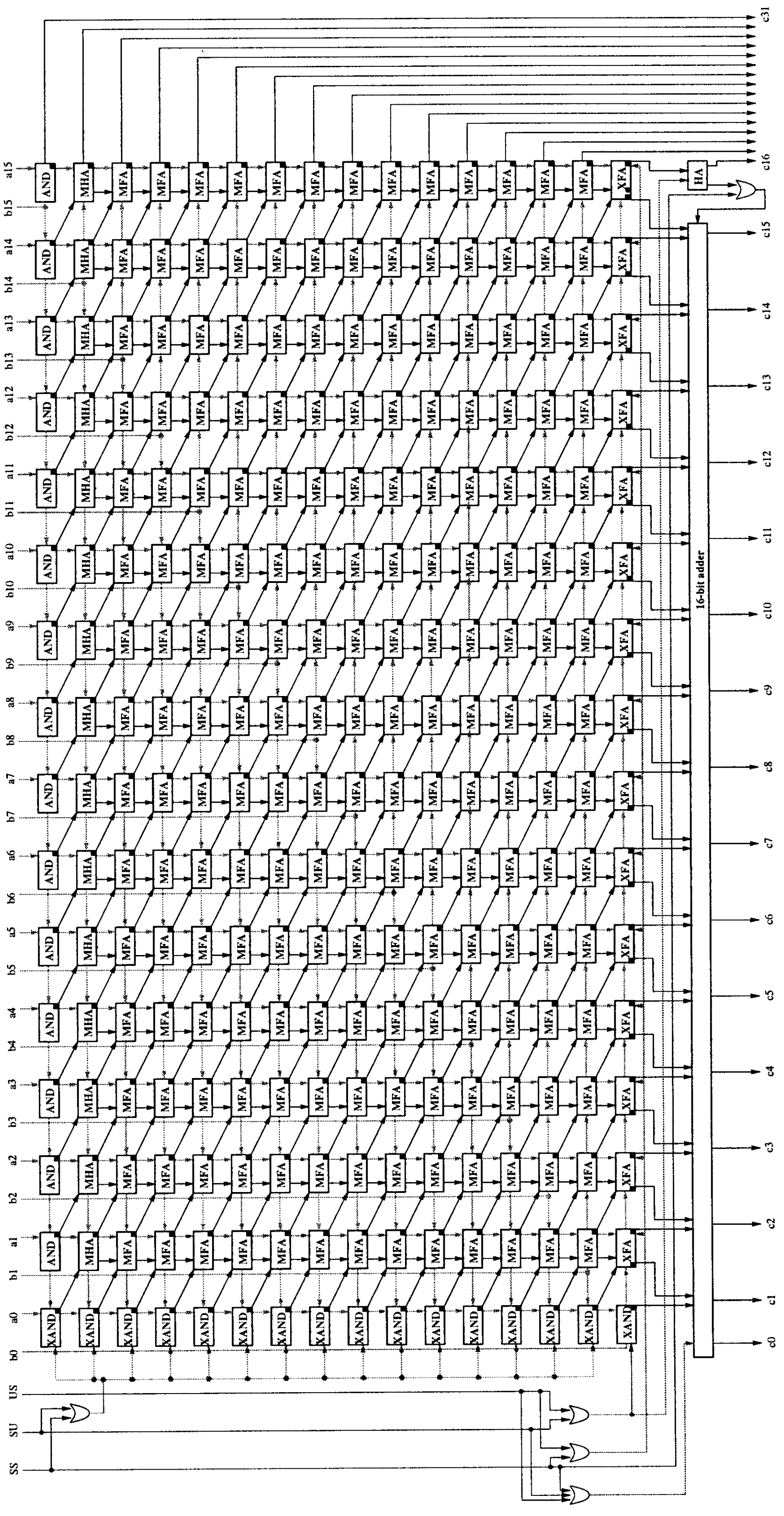


Figure 5.1: Block diagram of the 16-bit array multiplier with mixed-format operands.

For comparisons, the full adders are designed with three logic styles: complementary CMOS (Figure 3.1), SPL (Figure 3.3), and single-ended CPL-TG (Figure 3.6). The complementary CMOS full adder is readily available and parameterized in the cell library. The power-performance results and accurate parasitic information are also available for the array multiplier with dual-rail CPL-TG full adders (Figure 3.7), which was previously designed and fabricated.

5.2 Tree Multiplier Design

For potentially higher performance, a Wallace tree multiplier using 3-2 compressors is designed and modified based on the signed-unsigned algorithm. The tree format of the algorithm is illustrated in Figure 5.2. The block diagram is shown in Figure 5.3. The circuit is irregular, but can be designed to fit into 16-bit datapath. Multiplication is done in two stages, where the second stage is a 24-bit adder. The accumulation of partial products in the first stage is done in 6 logic levels.

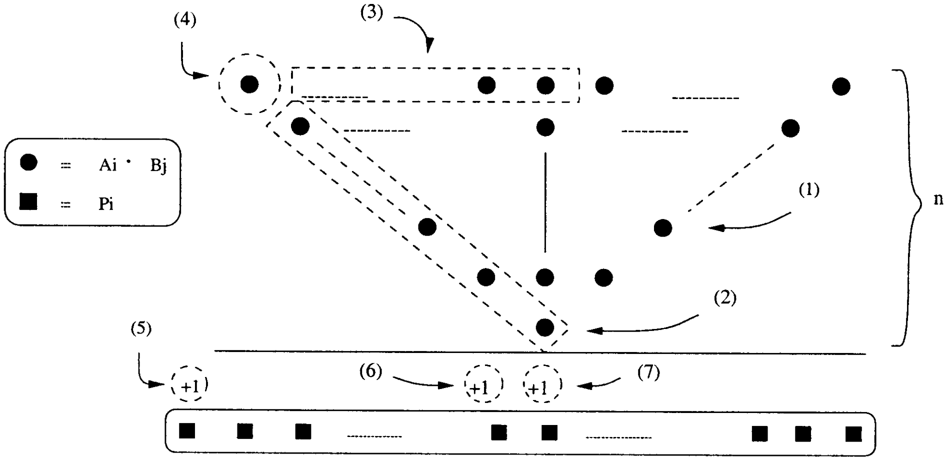


Figure 5.2: n-bit signed-unsigned multiplication: tree-format partial-product matrix.

For comparisons, the full adders are designed with three logic styles: complementary CMOS (Figure 3.1), SPL (Figure 3.3), and single-ended CPL-TG (Figure 3.6). Due to the tree multiplier's complex wiring scheme, the dual-rail CPL-TG full adder (Figure 3.7), which has complementary carry-out bits, are not selected.

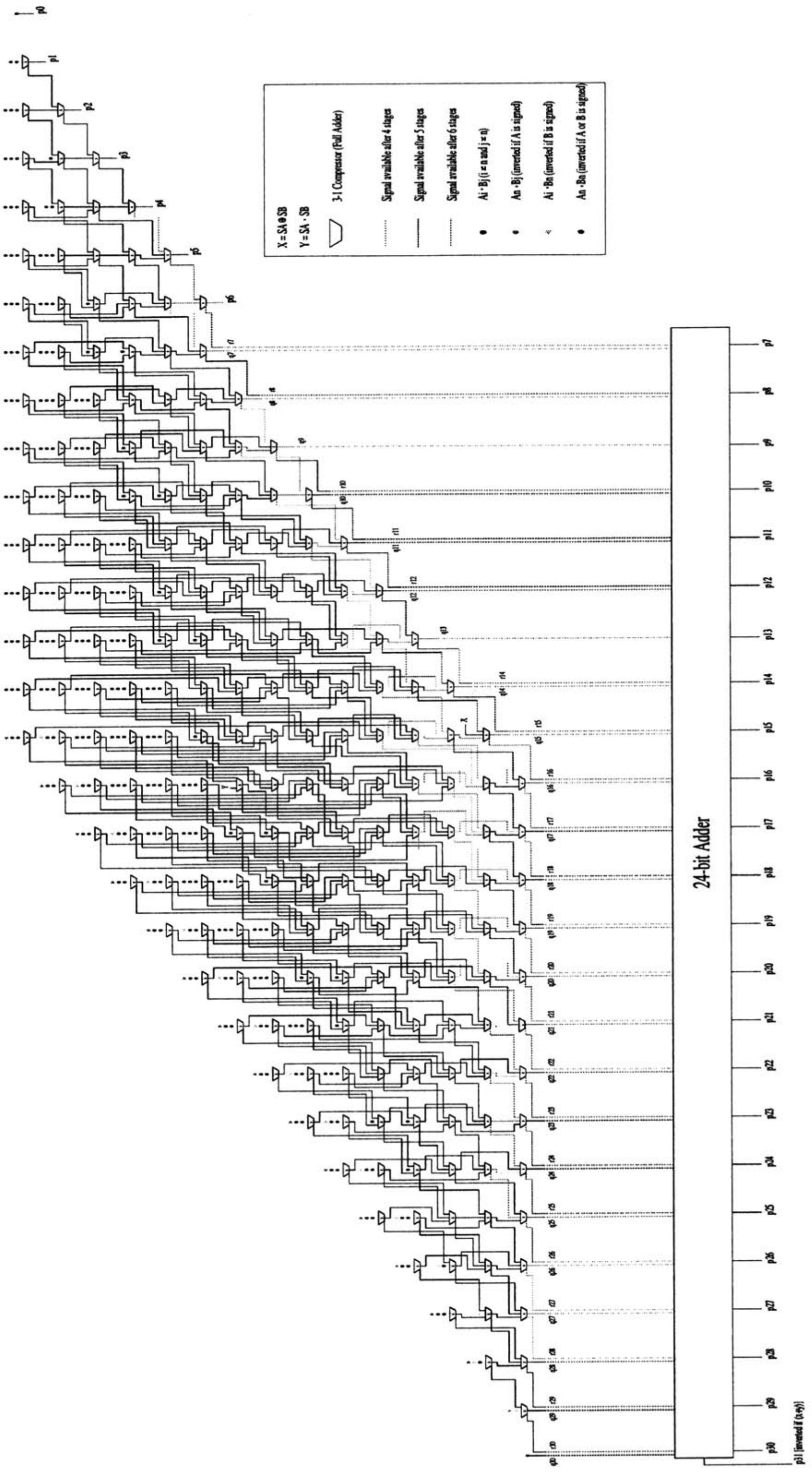


Figure 5.3: Block diagram of the 16-bit Wallace tree multiplier with mixed-format operands.

5.3 Simulation-Based Power and Critical-Path Delay Estimation

Power dissipation and critical-path delay of the multiplier can be accurately estimated using simulation-based techniques. We use a transistor-level power simulator and a static-timing tool called PowerMill and PathMill.

5.3.1 Transistor-Level Power Simulator

PowerMill is a transistor-level power simulator and analysis tool for CMOS circuit designs. It can run SPICE-accuracy power simulations with a run time considerably faster than SPICE by applying an event-driven timing simulation algorithm. Random vector inputs are generated and included as a stimulus file. For our design, the power simulation is running at 100 MHz for 100 switching cycles. The result is measured as an average energy dissipated per switching cycle.

5.3.2 Static-Timing Tool

PathMill is a transistor-level static-timing tool for custom circuit designs. While dynamic simulation techniques require user-defined vectors in order to simulate the critical path, PathMill finds critical delay paths based on topology, allowing automatic full coverage of the design. It searches all possible paths between endpoints in the design, and traces mutually exclusive, inverse, and other logical relationships to eliminate false paths.

Custom circuit topologies and timing checks are manually created and adjusted for some complex circuits such as cross-coupled PMOS devices and transmission-gate structures so that the timing results are matched with the results from dynamic-timing simulator. The timing verification process, as shown in Table 5.1 for example, is done by extracting critical delay paths from PathMill and reproducing them on PowerSPICE, which is a dynamic-timing simulator.

The timing results from PathMill are also verified against two other IBM-internal static-timing tools. All three timing tools give reliable results. Table 5.2 shows critical-path delays from PowerSPICE at various design corners. The performance can decrease by 10-40% on the worst-case corners.

Cell	PowerSPICE (delay in ns)	Pathmill (th=0.3)	Diff (%)	Pathmill (th=0.4)	Diff (%)	Pathmill (th=0.5)	Diff (%)
I2565(buf1)	0.084243	0.107	+27.0	0.107	+27.0	0.107	+27.0
I2529(buf2)	0.03855	0.038	-1.43	0.038	-1.43	0.038	-1.43
I1710(MHA)	0.13245	0.120	-9.40	0.120	-9.40	0.120	-9.40
I1907(FA)	0.11336	0.108	-4.72	0.112	-1.20	0.118	+4.09
I1911(FA)	0.11801	0.110	-5.09	0.114	-3.40	0.121	+2.53
I1910(FA)	0.11803	0.112	-5.11	0.114	-3.41	0.121	+2.52
I1909(FA)	0.11801	0.112	-5.09	0.114	-3.40	0.121	+2.53
I1908(FA)	0.11800	0.112	-5.08	0.114	-3.39	0.121	+2.54
I1913(FA)	0.11802	0.112	-5.10	0.114	-3.41	0.121	+2.52
I1912(FA)	0.11802	0.112	-5.10	0.114	-3.41	0.121	+2.52
I1906(FA)	0.11799	0.112	-5.08	0.114	-3.38	0.121	+2.55
I1905(FA)	0.11802	0.112	-5.10	0.114	-3.41	0.121	+2.52
I1902(FA)	0.11802	0.112	-5.10	0.114	-3.41	0.121	+2.52
I1901(FA)	0.11799	0.112	-5.08	0.114	-3.38	0.121	+2.55
I1903(FA)	0.11803	0.112	-5.11	0.114	-3.41	0.121	+2.52
I1904(FA)	0.13101	0.124	-5.35	0.125	-4.59	0.132	+0.76
I1711(XFA)	0.11958	0.138	+15.4	0.135	+12.9	0.133	+11.2
TOTAL	1.91730	1.865	-2.73	1.891	-1.37	1.979	+3.21

Table 5.1: Pathmill-PowerSPICE timing verification. Critical-path delays through the array multiplier at different relative thresholds of cross-coupled PMOS topology.

VDD	1.5V				1.4V			
Temp	50c		125c		50c		125c	
NRN	0.5 (NC)	0.87	0.5	0.87	0.5	0.87	0.5	0.87 (WC)
I2565	0.084243	0.094941	0.085508	0.096174	0.088238	0.100010	0.089108	0.100750
I2529	0.038558	0.045719	0.041887	0.049738	0.042390	0.050513	0.045675	0.054406
I1710	0.132490	0.160000	0.141060	0.170560	0.146110	0.177090	0.154660	0.187640
I1907	0.113350	0.137370	0.122350	0.148930	0.125370	0.152320	0.134490	0.164110
I1911	0.118010	0.142210	0.128330	0.154920	0.130220	0.157480	0.140670	0.170350
I1910	0.118030	0.142180	0.128340	0.154930	0.130230	0.157410	0.140650	0.170330
I1909	0.118020	0.142180	0.128320	0.154910	0.130210	0.157440	0.140640	0.170330
I1908	0.118000	0.142170	0.128330	0.154890	0.130210	0.157410	0.140650	0.170310
I1913	0.118020	0.142170	0.128320	0.154900	0.130210	0.157400	0.140650	0.170330
I1912	0.118020	0.142160	0.128320	0.154900	0.130210	0.157410	0.140630	0.170290
I1906	0.118000	0.142170	0.128320	0.154900	0.130200	0.157410	0.140630	0.170310
I1905	0.118020	0.142160	0.128320	0.154900	0.130200	0.157400	0.140630	0.170300
I1902	0.118020	0.142170	0.128320	0.154900	0.130210	0.157400	0.140640	0.170300
I1901	0.117990	0.142150	0.128320	0.154900	0.130190	0.157400	0.140640	0.170300
I1903	0.118020	0.142170	0.128320	0.154900	0.130200	0.157400	0.140620	0.170310
I1904	0.131590	0.158110	0.141690	0.170430	0.145090	0.174930	0.155310	0.187430
I1711	0.119550	0.147090	0.129180	0.159170	0.132950	0.164280	0.142530	0.176240
TOTAL	1.917900	2.307100	2.073200	2.498900	2.112400	2.550700	2.268800	2.744000
TOTAL%	+0	+20.29	+8.10	+30.29	+10.14	+32.99	+18.30	+43.07

Table 5.2: PowerSPICE results of critical-path delays at various corners

Chapter 6

Power-Performance Optimization

There are two main methods of circuit optimization in literature, dynamic tuning and static tuning [3]. The dynamic tuning is based on dynamic-timing simulation of the circuit, while the static tuning employs static-timing analysis to evaluate the performance of the circuit. This chapter explores some approaches for optimizing the 16-bit multiplier using both dynamic and static tuning tools.

6.1 Dynamic Tuning

The main advantage of dynamic tuning is its accuracy due to its realistic dynamic-timing simulation. However, dynamic tuning tool requires users to provide input patterns and a specific tuning problem in order to accurately optimize the circuit for all possible paths through the logic. Consequently, dynamic tuning is often applied only to small circuits in which the critical paths and the input patterns are easy to obtain.

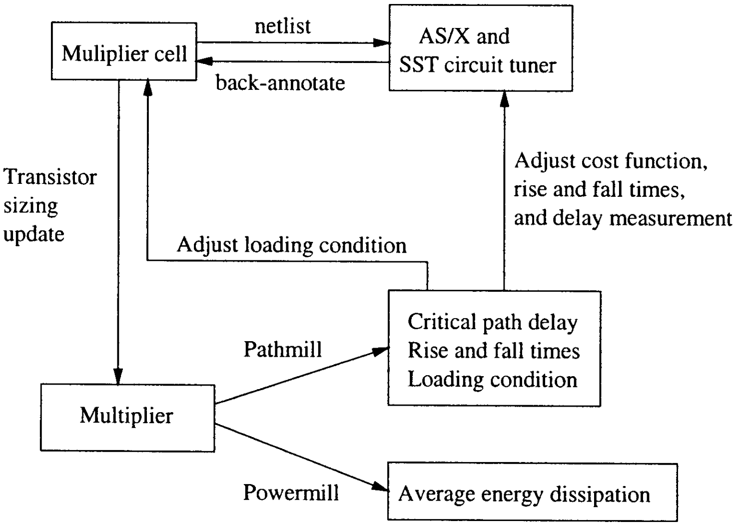


Figure 6.1: Dynamic tuning of a multiplier cell: process flow.

Although a 16-bit multiplier is too big to be tuned by a dynamic tuning tool, it is composed of multiple of identical multiplier cells that can be tuned in isolation with appropriate tuning conditions. As shown in Figure 6.1, the multiplier cell is tuned using an automatic tuning tool SST running on top of the SPICE-like circuit simulator AS/X. The multiplier cell is updated with new transistor sizes after the tuning and the multiplier is simulated and analyzed with PathMill and PowerMill. By examining the critical-delay paths through the multiplier reported by PathMill, the dynamic tuning setup can then be adjusted with more accurate loading conditions, rise/fall time, and input patterns.

Tuning Setup

The SST circuit tuner is set up for gradient-based cost-function minimization. The cost function is formulated such that its first derivative is continuous and the function will have its minimum value at the desired tuned state of the circuit. For an energy-efficient state at a given delay, we formulate the cost function as the following:

$$f = \alpha\left(\frac{delay}{delay_{\min}}\right)^2 + (1-\alpha)\left(\frac{energy}{energy_{\min}}\right)^2 \tag{6.1}$$

where α is an energy-delay coefficient, a real number between 0 and 1. The *delay* and *energy* variables are normalized by the coefficients $delay_{min}$ and $energy_{min}$.

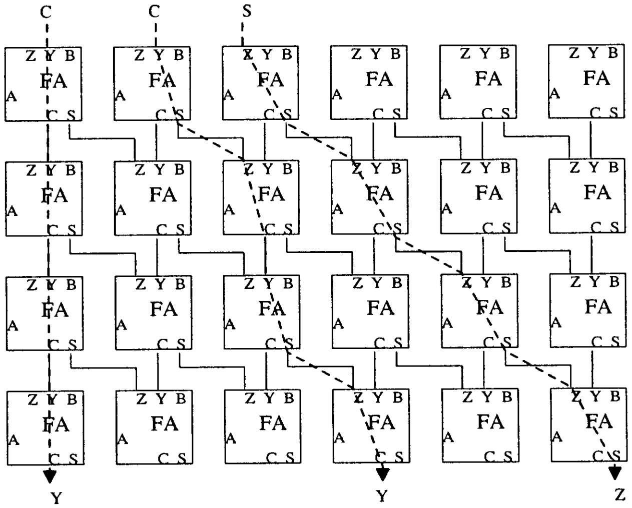


Figure 6.2: Critical-delay paths through an array multiplier

For an array multiplier, a multiplier cell with balanced carry-out and sum delays is desirable because both signals are on the critical paths. Consider possible critical-delay paths through the multiplier array shown in Figure 6.2. Inputs *A* and *B* of each FA are the operand bits for computing the corresponding partial product. Since all partial products will be computed simultaneously at the beginning, we will only consider the propagation delays through the paths from inputs *Y* and *Z* to outputs *C* and *S*. The input patterns are specified such that propagation delays through each path can be measured, and the final *delay* variable is computed as the following:

$$delay = \sqrt{\left(\frac{1}{4}\right) * \left(\tau_{y-c}^2 + \tau_{y-s}^2 + \tau_{z-c}^2 + \tau_{z-s}^2\right)} \qquad (6.2)$$

where $\tau_{i\text{-}j}$ is the propagation delay through the path from input *i* to output *j*.

The *energy* variable, which is the total energy dissipated per switching cycle, is computed as:

$$energy = E_{Vdd} + E_A + E_B + E_Y + E_Z \qquad (6.3)$$

where E_{Vdd} is the energy dissipated per switching cycle through the power supply and E_n is the energy dissipated per switching cycle through the input node *n*. They are computed as:

$$E_{Vdd} = \int_{cycle} [V_{dd} i_{Vdd}(t)] dt \qquad (6.4)$$

$$E_n = \int_{cycle} [(\frac{1}{2}) V_{dd} i_n(t)] dt \qquad (6.5)$$

Assume that the average voltage at the input node *n* is about one half of V_{dd} during switching.

6.2 Static Tuning

By utilizing EinsTuner, which is a gradient-based static-timing optimization tool [3], the whole multiplier can be tuned directly. Unlike dynamic-tuning tools, EinsTuner can efficiently handle large designs. All paths through the logic are simultaneously tuned and no input pattern is required. However, the utility of this tool is limited to the circuit that consists of pre-characterized library cells. In such case, transistor sizes can be optimized and updated directly, and layouts are easily generated.

For a full-custom 16-bit multiplier design, it is not practical to manually adjust all transistor sizes, and create schematics and layouts for all individual multiplier cells. In this project, we explore one way to address this problem by binding together the cells with similar transistor sizes.

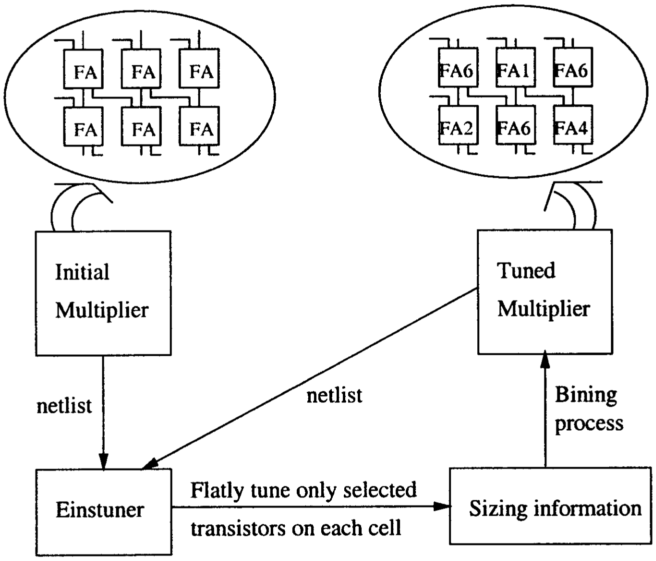


Figure 6.3: Static tuning of the multiplier: process flow.

The tuning process is illustrated in Figure 6.3. All multiplier cells are initially tuned by the dynamic tuner. To reduce the static tuning and binding workload, only a selected number of transistors inside each multiplier cell are tuned. EinsTuner flatly tune the multiplier for a higher speed until the area constraint is reached. Then, the transistor sizing information from EinsTuner is processed by a binding program, and the results are used to generate a new multiplier schematic.

Binding Algorithm

The maximum transistor size is limited to 1.2μm so that the area constraint is not exceeded. After the tuning, the EinsTuner output file is processed such that the multiplier cells whose all tuned transistor sizes are similar will be bound together. See the detailed process in Appendix B. Consider a case when only one transistor in each multiplier cell is tuned. Assume binding parameters k_1 and k_2 where 0.92μm $\geq k2 \geq k1 \geq$ 0. If the tuning range is less than or equal to k_1, the size of the tuned transistor is set to the mean value in all multiplier cells. If the tuning range is between k_1 and k_2, the transistor sizes are separated into two groups.

Otherwise, the transistor sizes are separated into three groups. Figure 6.4 shows an example of binding process with two transistors tuned, W1 and W2. Assume we set $k_1 = 0.1$ and $k_2 = 0.5$. If $0.5 \geq (W1_{max} - W1_{min}) > 0.1$ and $(W2_{max} - W2_{min}) > 0.5$, six new multiplier cells with different combinations of W1 and W2 will be created: $(W1_1,W2_1)$, $(W1_1,W2_2)$, $(W1_1,W2_3)$, $(W1_2,W2_1)$, $(W1_2,W2_2)$, and $(W1_2,W2_3)$, where

$$\begin{aligned} W1_1 &= (\frac{1}{4})[W2_{max} - W2_{min}] + W2_{min} \\ W1_2 &= (\frac{3}{4})[W2_{max} - W2_{min}] + W2_{min} \\ W2_1 &= (\frac{1}{6})[W2_{max} - W2_{min}] + W2_{min} \\ W2_2 &= (\frac{1}{2})[W2_{max} + W2_{min}] \\ W2_3 &= (\frac{5}{6})[W2_{max} - W2_{min}] + W2_{min} \end{aligned} \tag{6.6}$$

The multiplier cells will be lumped into one of these new multiplier cells according to the sizes of their tuned transistors.

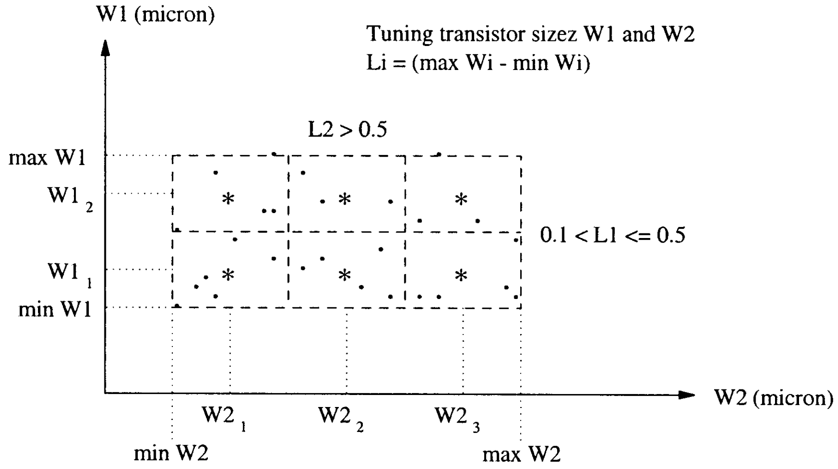


Figure 6.4: An example of binding process with two tuned transistors.

6.3 Energy-Efficient Curves

Several multiplier circuits are optimized using the two tuning methods explained above, and their energy-delay results are plotted. Then, an energy-efficient curve is constructed for each circuit, as illustrated in Figure 6.5, so that the impacts of multiplication structures and logic styles on the energy-delay characteristic of the multiplier can be easily compared. Each energy-efficient curve represents transistor sizes that provide the minimum energy dissipation for any given speed for the particular multiplier circuit configuration.

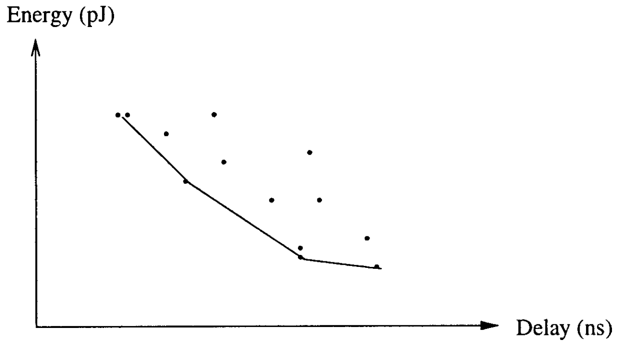


Figure 6.5: An energy-efficient curve represents transistor sizes that provide the minimum energy dissipation for any given speed.

Chapter 7

Results and Discussion

Several tree and array multipliers have been designed using various logic styles in 0.13μm CMOS technology. The circuits are optimized and simulated at two supply voltages, 1.5V and 0.9V. The resulting energy-delay plots are shown in Figure 7.1-7.4. In overall, the 16-bit array multiplier performs efficiently for a slower design, while the tree multiplier has a power-performance advantage for a faster design. The layouts of both multipliers, which can be fitted into a 16-bit datapath, are shown in Appendix A.

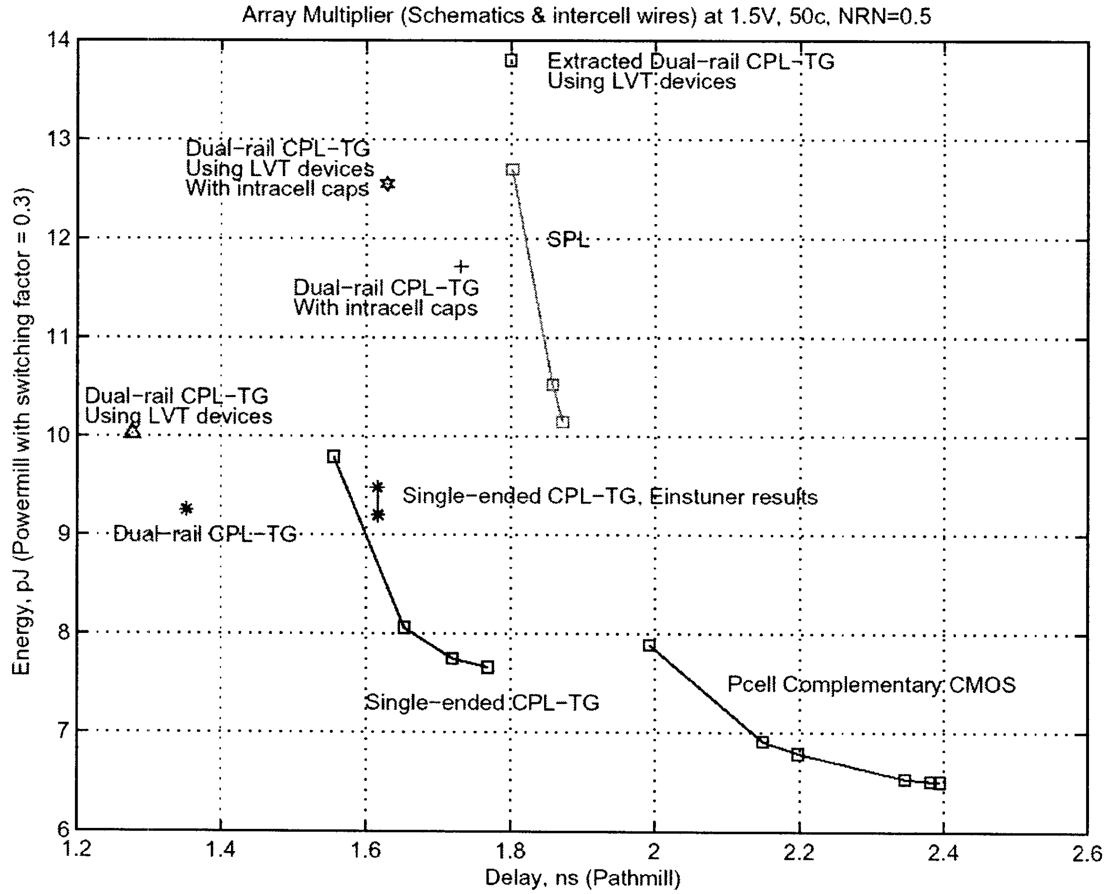


Figure 7.1: Energy-delay plots of array multipliers at 1.5V.

Only the single-ended CPL-TG array multiplier at 1.5V is optimized directly with the static tuner (EinsTuner) and the binding process. A detailed description of binding process can be found in section 6.2 and Appendix B. The results in Figure 7.1 show that this optimization approach does not yield any improvement. The binding process has proved too rough to get accurate results, and EinsTuner will only be useful if the multiplier cells are pre-characterized. For a full-custom design, the multiplier can be efficiently optimized by tuning its multiplier cell in isolation with dynamic tuning technique.

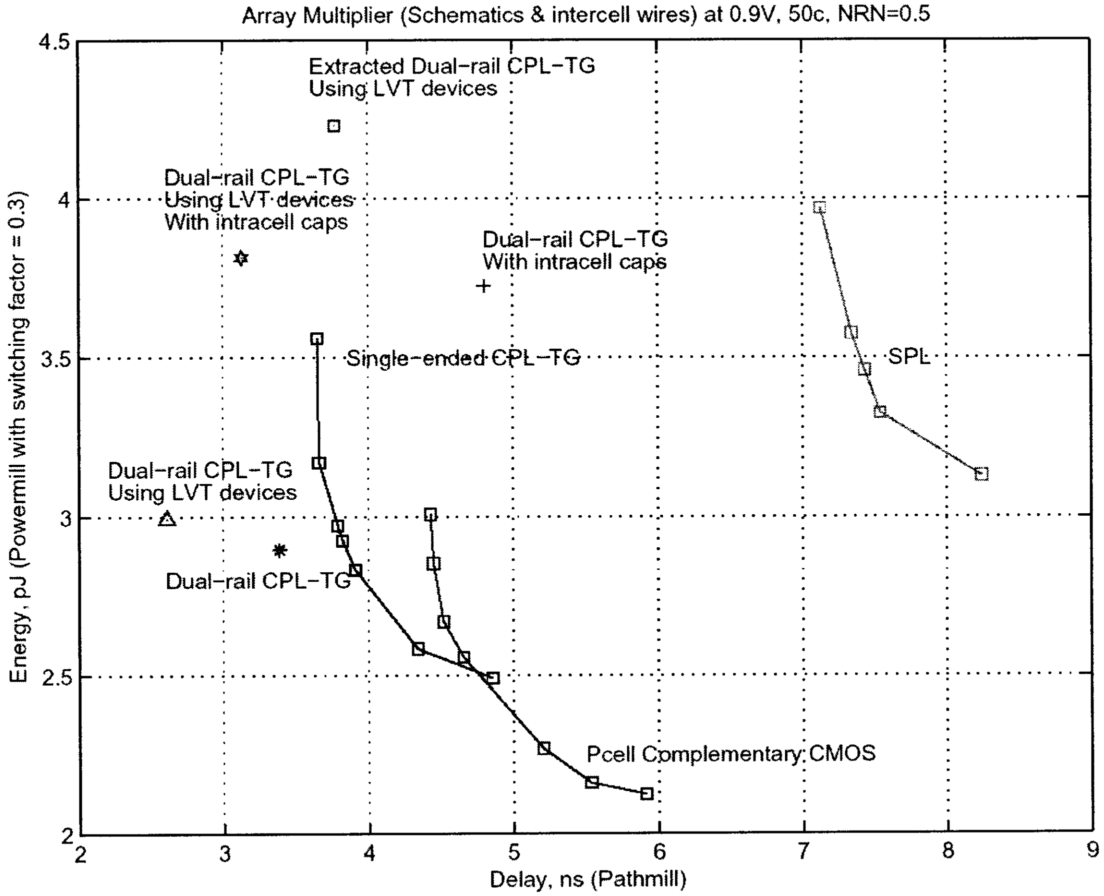


Figure 7.2: Energy-delay plots of array multipliers at 0.9V.

The results in Figure 7.1 and 7.2 also show that the dual-rail CPL-TG logic style has the best performance for array multiplier. Though, both delay and energy dissipation increase by 15-40% with more accurate parasitic information from layout extraction. The same effect will be expected for the other multipliers after their layouts are available. The dual-rail CPL-TG logic has a disadvantage of double wiring load due to its complementary outputs, and it is not a good choice for tree multiplier, which has an irregular structure and complex wiring scheme.

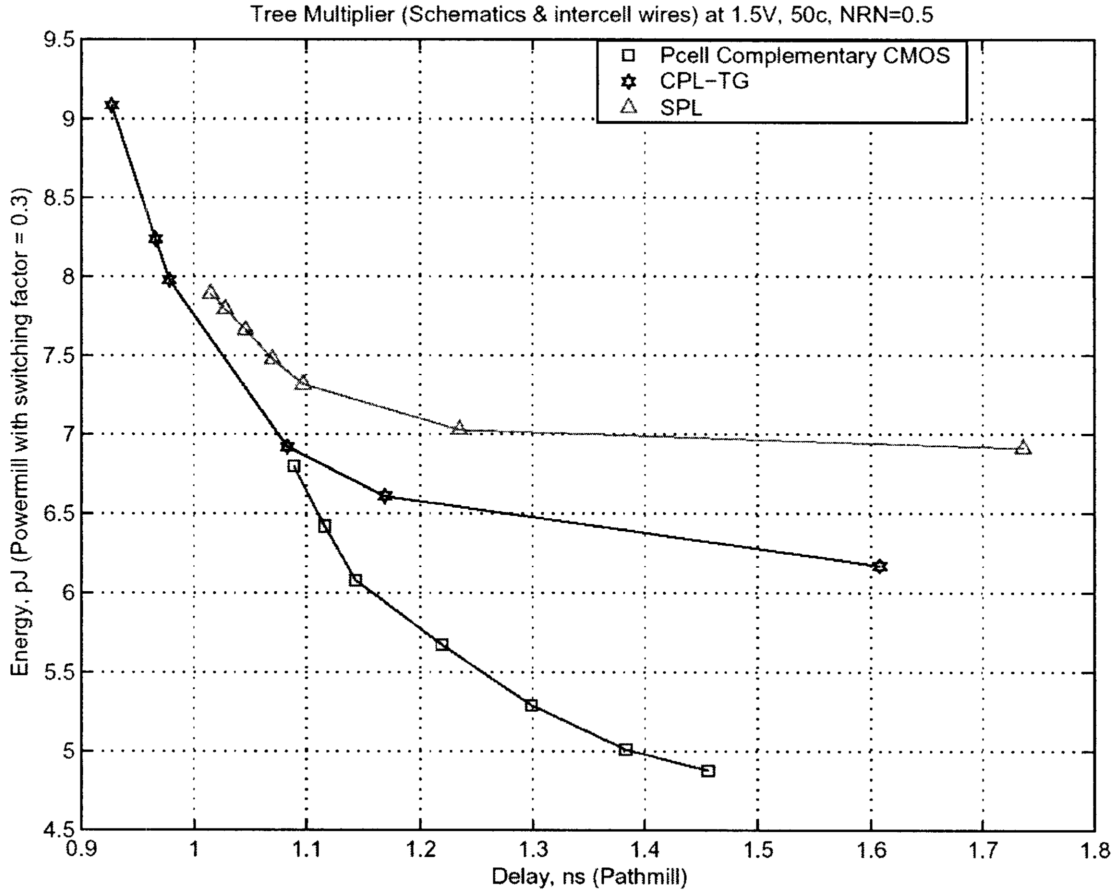


Figure 7.3: Energy-delay plots of tree multipliers at 1.5V.

Both single-ended CPL-TG logic and complementary CMOS logic are promising candidates for power-efficient design, in both array and tree multipliers. As the power supply is reduced from 1.5V to 0.9V, both logic styles also become even more efficient relative to SPL logic style. For SPL logic, the reduction of supply voltage from 1.5V to 0.9 increases the delay by 305% in array multipliers and 245% in tree multipliers, and reduces the power dissipation by 69% in array multipliers and 67% in tree multipliers. While, for single-ended CPL-TG and complementary CMOS, the reduction of supply voltage from 1.5V to

0.9V increases the delay by 120-140% in array multipliers and 133-150% in tree multipliers, and reduces the power dissipation by 65-66% in array multipliers and 66-67% in tree multipliers.

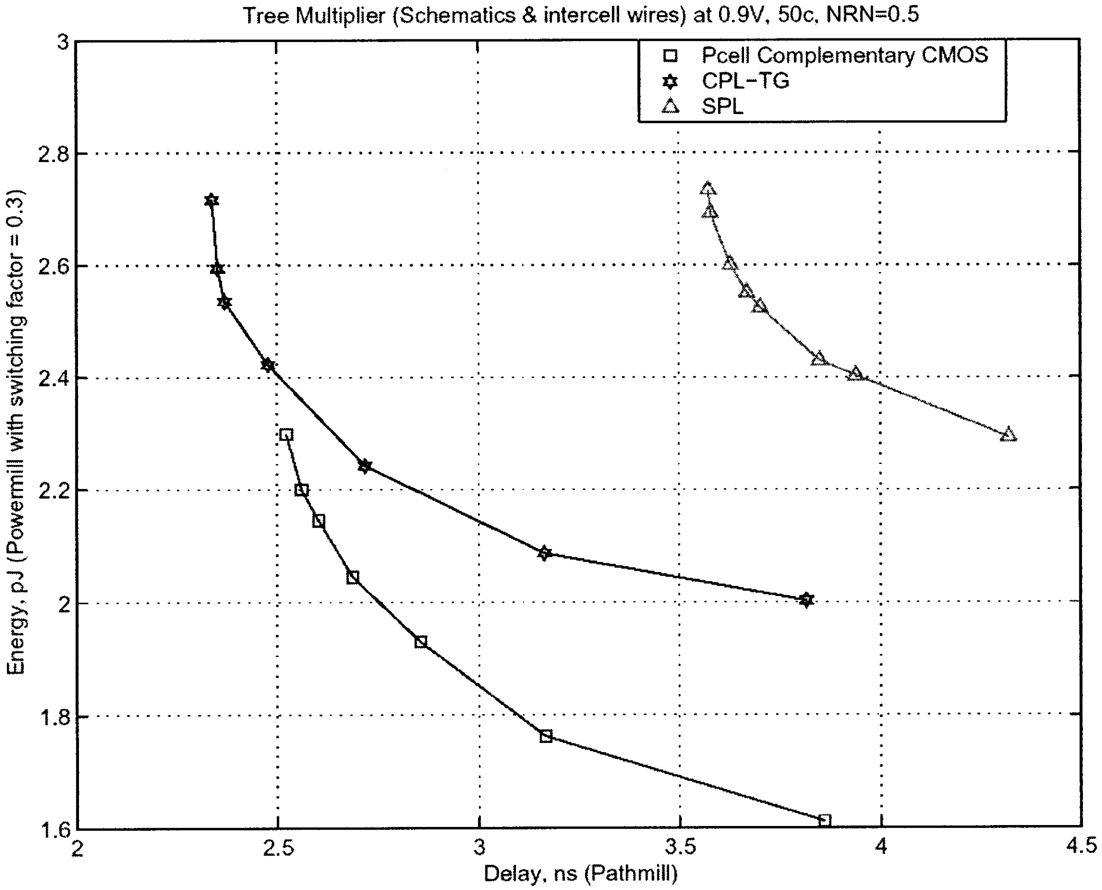


Figure 7.4: Energy-delay plots of array multipliers at 0.9V.

In comparison, complementary CMOS logic has power-performance advantage in a slower design. However, the single-ended CPL-TG logic becomes more efficient for a more aggressive design (faster speed).

Chapter 8

Conclusions

Several 16-bit array and tree multipliers have been successfully designed and implemented in 0.13μm CMOS technology at the supply voltage of 1.5V and 0.9V, using the proposed signed-unsigned algorithm. The multipliers are, then, used as a vehicle for exploring methodologies for power-efficient design at circuit level, specifically power reduction effects, transistor sizing using dynamic and static tuners, comparisons of circuit logic styles, and multiplication algorithms. The results yield interesting conclusions as the followings:

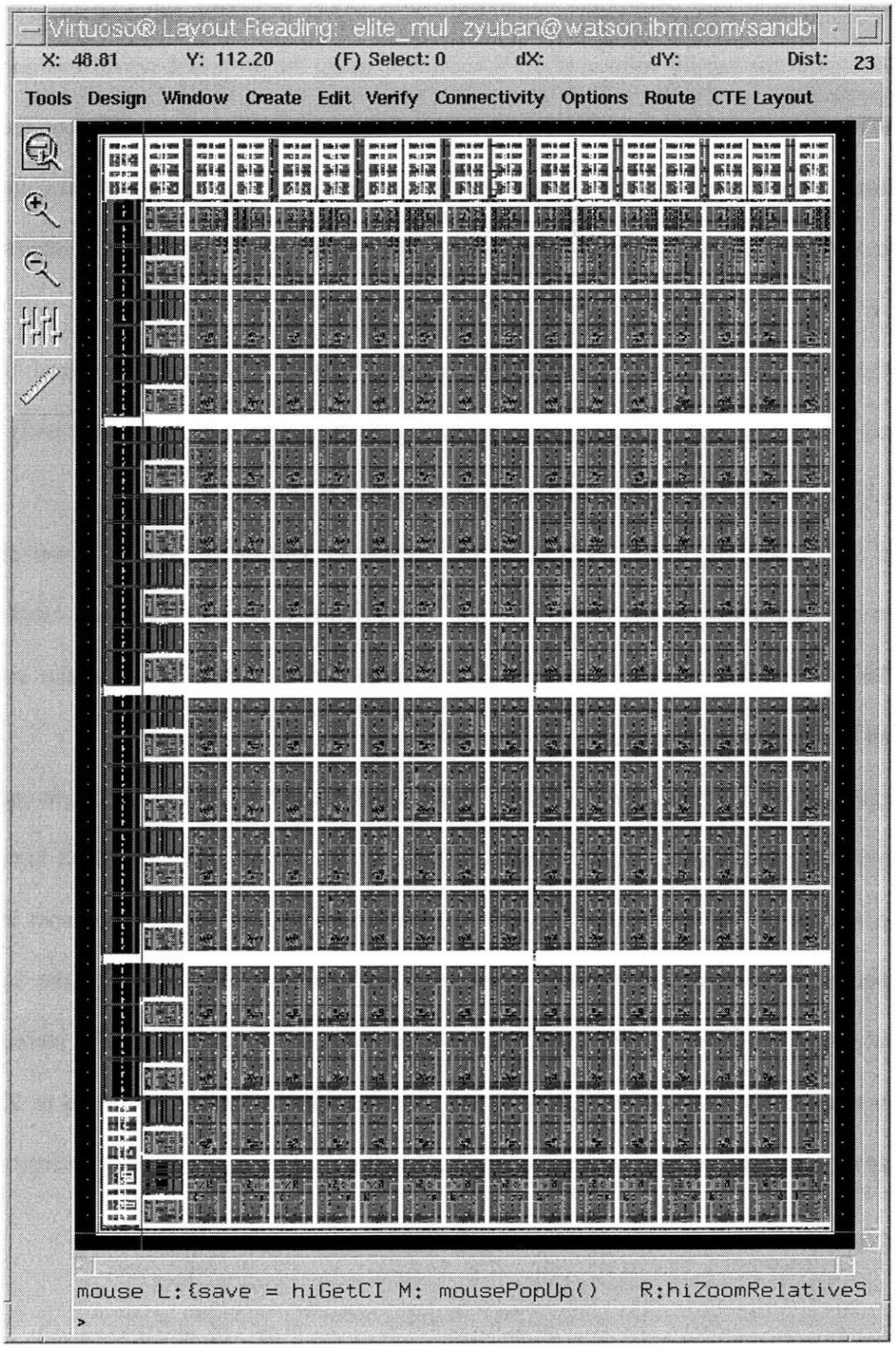
For a full-custom design, the multiplier circuits can be efficiently optimized by tuning its multiplier cell in isolation with dynamic tuning technique. Static tuners, which can efficiently handle large designs, will be useful only if pre-characterized multiplier cells are available.

The 16-bit array multiplier can be easily designed and optimized to perform power-efficiently due to its regular structure and wiring scheme. However, for a tighter delay constraint, the carefully-designed tree multiplier is a feasible solution despite its complicated wiring scheme and irregular structure. Both structures can be designed to fit into a 16-bit datapath.

Regardless of the multiplier structure, both complementary CMOS and single-ended CPL-TG logics are promising candidates for power-efficient design. The complementary CMOS logic has power-performance advantage in a slower design, while the single-ended CPL-TG logic becomes more efficient for a high-performance design. Both logic styles also perform relatively well with the supply voltage reduction technique, compared to SPL logic. The voltage reduction from 1.5V to 0.9V increases the delay by 140% on average for single-ended CPL-TG and complementary CMOS, compared to 275% for SPL logic style. In all cases, the voltage reduction from 1.5V to 0.9V achieves 67% power savings on average.

Appendix A

Multiplier Layouts

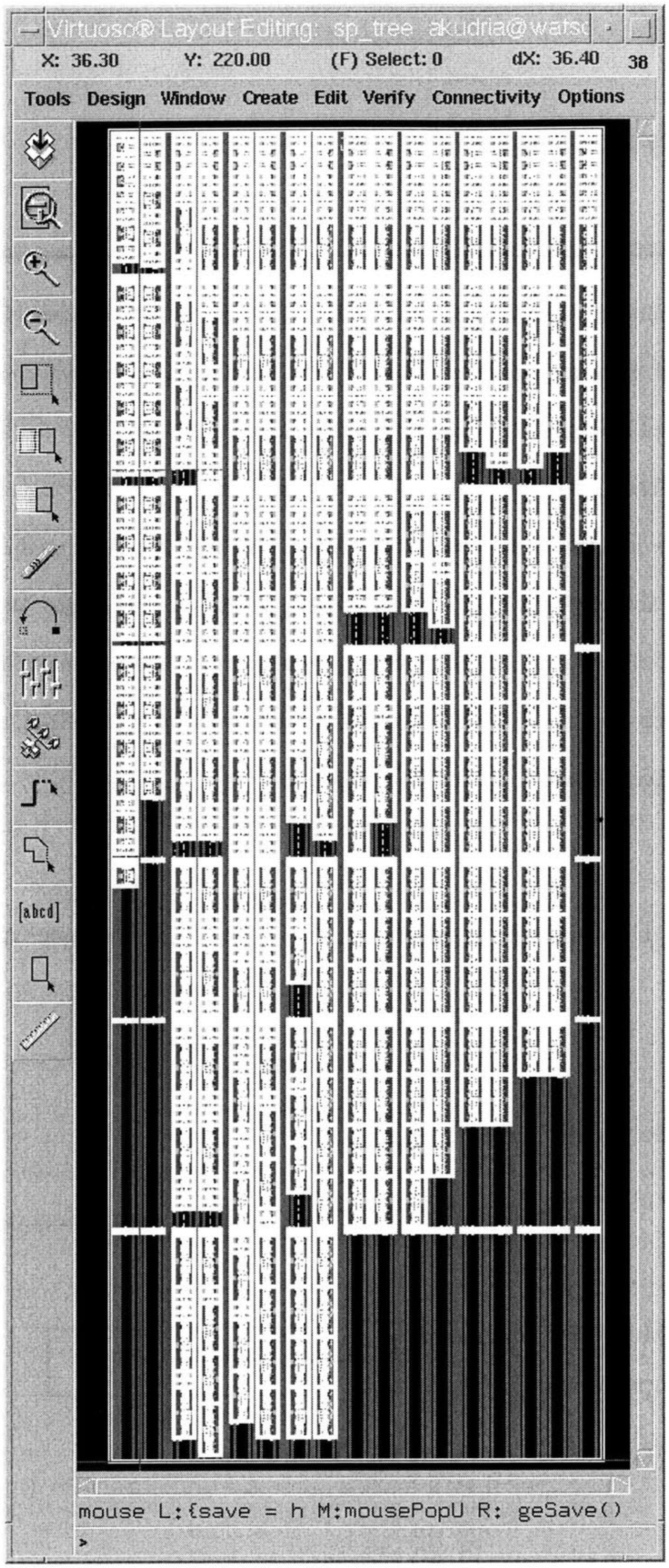


Virtuoso® Layout Reading: elite_mul zyuban@watson.ibm.com/sandb

X: 48.81 Y: 112.20 (F) Select: 0 dX: dY: Dist: 23

```
mouse L:{save = hiGetCI M: mousePopUp()    R:hiZoomRelativeS
>
```

Figure A.1: Array Multiplier Layout.



X: 36.30 Y: 220.00 (F) Select: 0 dX: 36.40 38

```
mouse L:{save = h M:mousePopU R: geSave()
>
```

Figure A.2: Tree Multiplier Layout.

Appendix B

Binding Programs

The following programs are used to process the output file from the static tuner (EinsTuner). Multiplier cells whose all tuned transistor sizes are similar will be bound together as explained in section 6.2. The process is in illustrated in Figure B.1. The main program *Binding.c*, which is written in C, takes three input files, *Instances.in*, *Fets.in*, and the output file from EinsTuner. *Instances.in* is a list of multiplier cell instances, and *Fets.in* is a list of transistors to be tuned. The program, then, produces three output files, *Schematic_type.out*, *Schematic_built.out*, and *Schematic_select.out*. The output files *Schematic_type.out* and *Schematic_built.out* are used by the user to manually create updated multiplier-cell schematics in Cadence. Then, the top-level multiplier schematic is updated by executing the SKILL file *Back_annotate.il* on Cadence, which takes *Schematic_select.out* as an input.

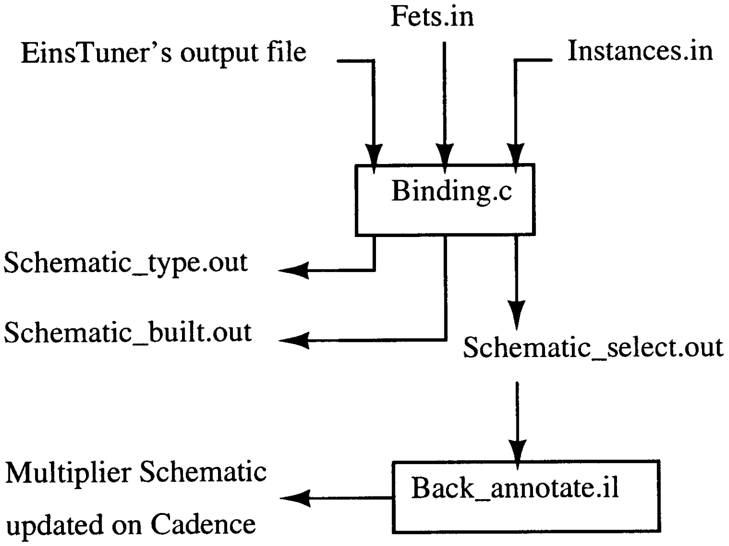


Figure B.1: Binding process.

Binding.c

```
/*** Format the Einstuner back-annotate output file ***/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct
{
  char name[4]; /*** i.e. P14, N18 ***/
  float width;
} fet;

typedef struct
{
  char name[7]; /*** i.e. I2_6, I10_14 ***/
  fet fets[10]; /*** #of fets in an instance to be tuned ***/
} instance;

typedef struct
{
  int num_block;          /*** 1,2 or 3***/
  float ref[2];     /*** reference points***/
  float width[3]; /*** widths to be assigned after comparison ***/
} fet_block;

main()
{
  /*** Declare variables***/
  instance instances[225];
  fet_block fets[10];
  int i,j,k,sch,fet_inc=0,num_sch=1,found;
  int num_fet=0, num_inst=0;
  int fet_pt[10]={0,0,0,0,0,0,0,0,0,0};
  int built[100], num_built=0;
  char *pch1, *pch2;
  char t1[25],t2[5],t3[10],t4[2],cell_name[15];
  char linei[100],tmp[10],inst_name[7],fet_name[4];
  char fet_list[10][4], inst_list[225][7];
  char inst_pat[225][10], sch_pat[7000][10];
  float wid;
  float fet_max[10]={0,0,0,0,0,0,0,0,0,0},
fet_min[10]={0,0,0,0,0,0,0,0,0,0};
  FILE *instList, *fetList, *schSelect, *schType, *schbuilt;

  /*** Open files***/
   instList = fopen("instances.in","r");
   fetList = fopen("fets.in","r");
   schSelect = fopen("schematic_select.out","w");
   schType = fopen("schematic_type.out","w");
   schbuilt = fopen("schematic_built.out","w");

   /*** Put instance and fet names in an array structure***/

   while (fscanf(fetList,"%s",&fet_list[num_fet]) != EOF)
```

```
       num_fet++;

   while (fscanf(instList,"%s",&inst_list[num_inst]) != EOF)
       num_inst++;

   for(i=0;i<num_inst;i++)
     {
       strcpy(instances[i].name,inst_list[i]);
       for(j=0;j<num_fet;j++)
       strcpy(instances[i].fets[j].name,fet_list[j]);
     }

   /*** Extract values from Einstuner file ***/
   while (fgets(linei,100,stdin) != NULL)
     {
       /*** read and extract a line of input ***/

sscanf(linei,"%s%3s%s%s%f%s%s",&t1,&t2,&tmp,&t3,&wid,&t4,&cell_name);
       pch1 = strtok(tmp,"/");
       pch2 = strtok(NULL,NULL);
       sscanf(pch1,"%s",&inst_name);
       sscanf(pch2,"%s",&fet_name);

       /*** store values into instance structure ***/
       /*** keep track of (min,max)width of each fet ***/
       i = 0;
       j = 0;
       while (strcmp(inst_name,instances[i].name) != 0)
       i++;
       while (strcmp(fet_name,instances[i].fets[j].name) != 0)
       j++;
       instances[i].fets[j].width = wid;
       if ((wid > fet_max[j]) || (fet_max[j] == 0))
         fet_max[j] = wid;
       if ((wid < fet_min[j]) || (fet_min[j] == 0))
         fet_min[j] = wid;
     }

   /*** Analyze data and generate schematic_type ***/
   for (j=0;j<num_fet;j++)
     {
       if ((fet_max[j]-fet_min[j]) <= 0.1)
       {
         fets[j].num_block = 1;
         fets[j].width[0] = ((fet_max[j]+fet_min[j])/2.0);
       }
       else if ((fet_max[j]-fet_min[j]) <= 0.3)
       {
         fets[j].num_block = 2;
         fets[j].ref[0] = ((fet_max[j]+fet_min[j])/2.0);
         fets[j].width[0] = (((fet_max[j]-fet_min[j])/4.0)+fet_min[j]);
         fets[j].width[1] = (((fet_max[j]-
fet_min[j])*(3.0/4.0))+fet_min[j]);
       }
       else
       {
         fets[j].num_block = 3;
```

```
         fets[j].ref[0] = (((fet_max[j]-fet_min[j])/3.0)+fet_min[j]);
         fets[j].ref[1] = (((fet_max[j]-
fet_min[j])*(2.0/3.0))+fet_min[j]);
         fets[j].width[0] = (((fet_max[j]-fet_min[j])/6.0)+fet_min[j]);
         fets[j].width[1] = ((fet_max[j]+fet_min[j])/2.0);
         fets[j].width[2] = (((fet_max[j]-
fet_min[j])*(5.0/6.0))+fet_min[j]);
       }
     }

   /*** number of schematics possible***/
   for (i=0;i<num_fet;i++)
     num_sch = (num_sch * fets[i].num_block);


   for (i=0;i<num_sch;i++)
     {
       fprintf(schType,"Schematic%d  ",i);
       for (j=0;j<num_fet;j++)
         {
         fprintf(schType,"%s=%5.2f
",fet_list[j],fets[j].width[fet_pt[j]]);
         if (fet_pt[j] == 0)
           sch_pat[i][j] = '0';
         else if (fet_pt[j] == 1)
           sch_pat[i][j] = '1';
         else
           sch_pat[i][j] = '2';
       }
       fprintf(schType,"\n");
       if (fet_pt[fet_inc] < (fets[fet_inc].num_block-1))
       fet_pt[fet_inc]++;
       else if (fet_inc < (num_fet-1))
       {
          while (fet_pt[fet_inc] == (fets[fet_inc].num_block-1))
          fet_inc++;
          fet_pt[fet_inc]++;
          for (j=0;j<fet_inc;j++)
          fet_pt[j] = 0;
          fet_inc = 0;
       }
     }

   /*** Compare widths and generate schematic_select***/
   for (i=0;i<num_inst;i++)
     {
       fprintf(schSelect,"%s %s",inst_list[i],cell_name);
       for (j=0;j<num_fet;j++)
       {
         if (fets[j].num_block == 1)
           {
             inst_pat[i][j]='0';
           }
         else if (fets[j].num_block == 2)
           {
             if (instances[i].fets[j].width <= fets[j].ref[0])
             inst_pat[i][j]='0';
```

```
              else
              inst_pat[i][j]='1';
            }
          else
            {
              if (instances[i].fets[j].width <= fets[j].ref[0])
              inst_pat[i][j]='0';
              else if (instances[i].fets[j].width <= fets[j].ref[1])
              inst_pat[i][j]='1';
              else
              inst_pat[i][j]='2';
            }
        }
        sch = 0;
        while ((strcmp(inst_pat[i],sch_pat[sch]) != 0) && (sch <
num_sch-1))
          sch++;
        fprintf(schSelect,"%d\n",sch);

        k = 0;
        found = 0;
        while ((k < num_built) && (found == 0))
        {
          if (sch == built[k])
            found = 1;
          else
            k++;
        }

        if (found == 0)
        {
          built[num_built] = sch;
          num_built++;
        }
      }

    /*** Print out numbers of schematics to be built ***/
    for (k=0;k<num_built;k++)
      {
        fprintf(schbuilt,"%d ",built[k]);
      }

    fclose(instList);
    fclose(fetList);
    fclose(schSelect);
    fclose(schType);
}
```

References

---

[1] J.H. Moreno, V. Zyuban, U.S. Shvadron, A.Z. Zaks, *Apparatus for Multiplication of Data in Two's Complement and Unsigned Magnitude Formats.* U.S. Patent Application No. US20040010536A1, July 11, 2002.

[2] V. Zyuban, P. Strenski, *Unified methodology for resolving power-performance tradeoffs at the microarchitectural and circuit levels.* In International Symposium on Low-Power Electronics and Design, pp. 166-171, August 2002.

[3] A. Conn, et al., *Gradient-based optimization of custom circuits using a static-timing formulation.* In Proceedings of Design Automation Conference, pages 452-459, June 1999.

[4] Muhammad S. Elrabaa, Issam S. Abu-Khater, and Mohamed I. Elmasry, *Advanced Low-Power Digital Circuit Techniques.* Kluwer Academic Publishers, 1997.

[5] Kerry Bernstein, Keith M. Carrig, et al., *High Speed CMOS Design Styles.* Kluwer Academic Publishers, 1998.

[6] J. M. Rabaey, A. Chandrakasan, and B. Nikolić, *Digital Integrated Circuits: A Design Perspective, second edition.* Prentice Hall Electronics and VLSI Series, 2003.

[7] T. Kao and A. Chandrakasan, *Dual-Threshold Voltage Techniques for Low-Power Circuits.* In IEEE Journal of Solid-State Circuits, vol. 35, no. 7, pp. 1009-1018, July 2000.

[8] D. Soudris, C. Piguet, and C. Goutis, *Designing CMOS Circuits for Low Power.* Kluwer Academic Publishers, 2002.

[9] K. Yano et al. *Top-Down Pass-Transistor Logic Design.* In IEEE Journal of Solid-State Circuits, Vol. 31, No. 6, June 1996.