A Stream-Based In-Line Allocatable Multiplier for Configurable Computing

by

Tsung-Han Yang

Thesis submitted to the faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

APPROVED:

Peter M. Athanas, Chairman

James R. Armstrong

Nathaniel J. Davis, IV

August 29, 1997

Blacksburg, Virginia

Keywords: VLSI, Multiplier, Computer Arithmetic

Copyright 1997, Tsung-Han Yang

A Stream-Based In-Line Allocatable Multiplier for Configurable Computing

by

Tsung-Han Yang

Committee Chairman: Peter M. Athanas

Electrical Engineering

(ABSTRACT)

The growing demand for high-performance computing platforms has pushed the computing community to invent new architectures for processors. Recently, researchers have begun to solve the problem by the implementation of Field-Programming Gate Arrays (FPGAs). FPGAs make it possible to implement different applications on the same hardware. Unfortunately, FPGAs suffer from low bandwidth, density, and throughout. To gain the flexibility of FPGAs and to gain more computational capacity than conventional processors have, Wormhole run-time reconfigurable (RTR) techniques has been developed to address some high performance digital signal processing (DSP) problems.

Multiplication is one of the basic functions used in digital signal processing. Most high-performance DSP systems rely on hardware multiplication to achieve high data throughput. To meet the processing needs of DSP, a multiplier was embedded into a prototype wormhole RTR device called Colt, but because each design has its own speed and size requirements, rarely can a designer take an already existing multiplier module and use it in Colt. Therefore redesigning multipliers is necessary for meeting the system specifications of Colt. This thesis explores the design of the multiplier from architecture level to circuit level.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Athanas, for giving me the opportunity to work on an exciting project in my area of interest and for his support and expertise throughout the project and my graduate career. I also like to thank Dr. Armstrong and Dr. Davis for taking their time to be on my committee. Thanks also to the other members of this project: Ray Bittner, Mark Cherbaka, Mark Musgrove and Brian Kahne.

Finally, thanks to my wife for her support and patience during these years. Without her, this thesis is not possible to be done at this time.

TABLE OF CONTENTS

1 Introduction **1**

1.1 Motivation 1

1.2 Methods 2

1.3 Contribution 3

1.4 Organization 3

2 Colt Architecture **5**

2.1 Overview of Colt Architecture 5

2.2 Crossbar and Data Port 8

2.3 Mesh 9

2.4 Interconnected Function Unit 10

2.5 Function Unit 11

2.6 Multiplier 13

3 Architecture of Multiplier **15**

3.1 Overview 15

3.2 Multiplier Architectures 16

3.2.1 Array Multiplier 17

3.2.2 Tree Multiplier 18
3.2.3 Booth Recoded Multiplier 21
3.3 Linear Pipeline 23
3.3.1 Synchronous Model 24
3.3.2 Clocking and Timing Control 24
3.3.3 Speedup 25
3.3.4 Pipelined Multiplier 26
3.4 Solution for the Architecture 26
4 Circuit Design **28**
4.1 CMOS Logic Structures 28
4.1.1 CMOS Complementary Logic 29
4.1.2 Pass-Transistor Logic 30
4.1.3 Complementary Pass-Transistor Logic (CPL) 30
4.2 Adder 32
4.2.1 Complementary CMOS Full Adder 34
4.2.2 Transmission-Gate Adder 34
4.2.3 Modified CPL Full Adder 37
4.2.4 Comparison of the Adders 38
4.3 Multiplexer 40
4.4 Multiplier Cell 40
4.5 Flip-Flop 42

4.6 Architecture 43

5 Floor Plan and Layout 46

5.1 Strategy 46

5.1.1 Regularity 46

5.1.2 Modularity 47

5.1.3 Locality 47

5.2 Floor Plan and Layout 47

5.3 Verification 49

6 Testing and Application 51

6.1 Testing Environment 51

6.1.1 Language 52

6.1.2 The Loader Program 52

6.1.3 Evaluation Board 53

6.1.4 Xilinx Configuration 54

6.2 Application: Matrix Multiplication 55

6.2.1 Overview 57

6.2.2 Data Flow Graph Mapping 58

7 Conclusion 63

7.1 Result 63

7.2 Future Enhancement 63

7.2.1 Signed Multiplication 64

7.2.2 Speed 65

7.2.3 Adder 66

References **68**

A HP05 Design Process **70**

A.1 MOSIS Parametric Measurements 70

A.2 MOSIS Design Rules 73

B Various Tools and Source Codes **75**

B.1 th2spi.awk 75

B.2 mx 76

B.3 The Loader Program 83

B.4 VHDL code of the Stream Controller 86

B.5 Tier1 Program of Matrix Multiplication 88

Vita **91**

LIST OF FIGURES

2.1 The Major Devices in the Colt Architecture. 6

2.2 An Illustration of the Stream Format. 8

2.3 The Mesh Topology within the Colt Chip. 9

2.4 Interconnected Functional Unit. 10

2.5 Functional Unit. 11

2.6 ALU Bit Slice. 12

3.1 A 4x4 Array Multiplier. 17

3.2 A 4x4 Tree Multiplier 20

3.3 A 16x16 Multiplier Top Level Schematic and Floor Plan 23

3.4 A Synchronous Pipeline Model. 24

4.1 (a) CMOS Inverter. (b) CMOS NOR Gate. 29

4.2 Two-input XNOR Gate Implemented in Pass-Transistor Logic. 30

4.3 CVSL Schematic Structure. 31

4.4 CPL Schematic Structure. 32

4.5 CPL AND/NAND Circuit. 33

4.6 CMOS Full Adder[Wes93] 35

4.7 Transmission Gate Adder with 18 Transistors [Wes93] 36

4.8 Full Adder Implemented by Transmission Gates 37

4.9 The Delay of 4-bit Modified CPL Adder and Transmission-Gate Adder. 38

4.10 The Delay of 16-bit Modified CPL Adder and Transmission-Gate Adder. 39

4.11 NMOS Multiplexer. 40

4.12 The Cell of the Multiplexer. 41

4.13 The AND Gate by Pass-Transistor Logic. 41

4.14 D Flip-Flop [Bit97a]. 42

4.15 The Delay of 32-bit modified CPL Adder. 43

4.16 The Block Diagram of the Multiplier. 44

5.1 Floor Plan of Multiplier. 48

5.2 Floor Plan of a Row. 48

5.3 The Layout of the Multiplier. 50

6.1 Testing Environment 51

6.2 The Block Diagram of Colt Evaluation Board 53

6.3 A Data Stream File 54

6.4 Tier1 Source of the Multiplication 56

6.5 Matrix Multiplication 58

6.6 Configuration for Matrix Multiplication 61

6.7 Data Stream Example 62

7.1 Implementation of Booth Multiplier 64

7.2 Speed Improvement of Multiplier Architecture 65

7.3 Comparison of Four Adders 66

LIST OF TABLES

2.1 Comparison of 16-bit Unsigned FPGA Multiplier Implementations. 13

3.1 4-bit Multiplier Partial Products 15

3.2 The Truth Table of a Full Adder 18

3.3 Booth Multiplier Recoding Table [Ham90] 22

3.4 The Multiplier and The Recoded Multiplier by Booth Algorithm[Ham90] 22

3.5 Booth Multiplier Bit-Pair Recoding Table [Ham90] 22

4.1 The Truth Table of a Full Adder [Wes93]. 33

4.2 Timing Property of D Flip-Flop [Che96]. 42

5.1 The Sheet Resistance of HP 0.5μ Process. 49

6.1 The ALU Term Configuration 58

7.1 Dimension and Current of the Multiplier. 63

Chapter 1

Introduction

Multiplication is one of the basic functions used in digital signal processing (DSP). Most high-performance digital signal processing systems rely on hardware multiplication to achieve high data throughput. The Colt integrated circuit is the computational core of a prototype hardware platform for performing high speed DSP, and will be called upon to do a large number of multiplications. The important point is how the multiplication is implemented. The design issue of the multiplier from circuit level to architecture level will be discussed and explored in this thesis.

1.1 Motivation

In the past, many novel ideas for multipliers have been proposed to achieve high performance. They fall into two general categories: "tree" multipliers and "array" multipliers. Tree multipliers add as many partial products in parallel as possible and therefore are very high performance architectures. Unfortunately, tree multipliers are also very irregular, hard to lay out and hence large. Array multipliers, on the other hand, are very regular, small in size, but suffer in latency and propagation delay [Ste90].

The goal of a practical design is not only high performance but also a reasonable price. Reasonable pricing should be considered from the aspect of both time and area, which means an easier

layout and a smaller design. The problem with contemporary FPGA-based multiplier designs is that they are too slow and too large. The goal of this research is to design an embedded multiplier in the Colt chip in a manner to make it readily accessible and allocatable, and to minimize the occupied area. Many evaluations and comparisons will be discussed in this thesis.

1.2 Methods

Several tools were used in the design approach of the multiplier. Cadence was used for the logic simulation and the layout of the multiplier. Cadence integrates the back-end and front-end phase of the design process. This tool provides a digital simulator and a schematic editor for the front end of the design process. For the back end, it contains a layout editor and a design rule checker (DRC). Layout-versus-schematic (LVS) check integrates the front and the back end. Spice3 is used for the analog simulation. AWK is used to generate the pattern of the simulation.

Every level of design has been carefully considered and verified to make sure the proposed architecture and circuit work. Extensive verification was done by computer-aided design (CAD) tools. Digital simulators were used to verify the concept and the logic of the multiplier. An analog simulator was used to optimize the performance of the circuit and to verify the operation of the circuit. Many errors have been removed by this method before fabrication of the integrated circuit.

Colt was constructed by MOSIS scalable CMOS technology with their HP05 process. This is a three-metal n-well process with a 0.6 μm drawn feature size (0.5 μm effective). Circuit geometries are drawn with a λ-based methodology [Wes93] that has a unit of measurement, $\lambda = 0.3$ μm.

1.3 Contribution

A multiplier is an essential element in any digital-signal processing (DSP) circuit and constitutes the critical path in DSP and FPU (floating-point unit) in VLSI. As the process of fabrication continues progressing, it is possible to embed a multiplier in an integrated circuit. Parallel multipliers are notorious for their resource-consuming implementation in FPGA-type architectures. It is a reasonable conclusion that a FPGA should embed a multiplier if it does a great amount of multiplications. Unfortunately, because of the limitations of contemporary FPGA architectures, a multiplier has never been successfully embedded in FPGAs.

This research describes a fast 50M Hz pipelined multiplier embedded in a wormhole run-time reconfigurable (RTR) integrated circuit. A new approach of integrating a multiplier into a RTR architecture has been developed. To optimize the trade-off between the area of the multiplier and the speed of the multiplication, several architectures of multiplier have been investigated. The circuit and architecture of the final multiplier of this research have the potential for 100 MHz operation of 16x16 multiplication on the same process.

1.4 Organization

This thesis is organized into six subsequent chapters and two appendixes. Chapter 2 presents background information on the Colt architecture necessary to understand the design and its operation. This includes individual sections on the major components in Colt, such as the data port and crossbar (Section 2.2), mesh (Section 2.3), interconnected function unit (Section 2.4), function unit (Section 2.5), and the multiplier (Section 2.6).

Chapter 3 investigates several options for the design of the multiplier. The first section (Section 3.1) gives an overview about the operation of the multiplier. Section 3.2 investigates versatile architectures of the multiplier such as the array multiplier, tree multiplier and Booth recoded multiplier. Section 3.3 discusses the concept of the pipeline and the implementation of a pipeline multiplier. Section 3.4 compares the architectures of the multiplier and discusses the benefit of the proposed architecture.

Chapter 4 explores deeper into the detail of the circuit design. Section 4.1 shows the options of the logic structures based on the HP $0.5 \mu m$ CMOS process. Conventional CMOS logic, pass-transistor logic and complementary pass-transistor logic are used in the design. Section 4.2 compares several schematics of full adder. The rest of the chapter discusses the other circuits of the multiplier. The last section (Section 4.6) describes the final design of the multiplier.

Chapter 5 discusses the issues of the layout. Section 5.1 shows the strategy while implementing layout. The details of the floor plan and layout are discussed on Section 5.2. The last section discusses the verification of the layout.

Chapter 6 is about the testing and applications of the multiplier. Chapter 7 shows the result of the design and discusses the future improvement of the multiplier.

Chapter 2

Colt Architecture

Since the multiplier is embedded in the Colt chip, it is essential to have some knowledge of Colt architecture. This chapter will explain enough of the architecture to understand this thesis. A more detailed description about Colt architecture can be found in [Bit97a] , [Che96], and [Mus96].

2.1 Overview of Colt Architecture

The Colt architecture can be categorized into a rather new division of computer hardware called custom computing machines (CCMs). These architectures promise to bring hardware-like performance with software-like configuration [Che96]. The dominant architecture of CCMs today is a class of chip called Field Programming Gate Array (FPGA) which uses Configurable Logic Blocks as processing units [Xil94]. However, all commercial FPGAs today can only be reconfigured statically. Colt has improvement on FPGAs by providing a dynamically and partially reconfigurable ability.

The types of computations that are involved with DSP type applications, such as the implementation of an FIR or Adaptive Filter, suggest a data flow approach to maximize the parallelism of the execution [Bit97a]. Hence, Colt adopts a data flow approach. Each function unit can be configured to perform specific arithmetic operation and to route the result to another unit. For

practical consideration, a mesh network is used as the interconnected network to reduce the chip size.

The core of the Colt architecture consists of a crossbar, six data ports and a mesh of processing elements as shown in Figure 2.1. The mesh of the processing elements can be divided into 16 interconnected functional units (IFU) and then into functional units (FUs) [Mus96]. Data flows into the data ports and passes through the crossbar, where it is routed to the mesh. Most elements of the Colt are reconfigurable.

Colt is a wormhole RTR prototype. In wormhole RTR, a stream is the basic unit of computation. A stream is a concatenation of a programming header and oprand data as shown Figure 2.2. The programming header is used to configure a computational pathway through the system as well as to configure the operations to be performed by the various computational elements along the path. The stream is self-steering and, as it propagates through the system, configuration information is stripped from the front of the header and is used to program the unit at the head of the stream; thus, the size of the header diminishes as the stream propagates through the system. The stream header is composed of an arbitrary number of packets of programming information. Each packet contains all of the information needed to configure a designated unit in the system. The composition and length of the packets are variable so that different packet types may coexist within the same stream header and hence heterogeneous unit types may be traversed by a given stream [Bit97b].

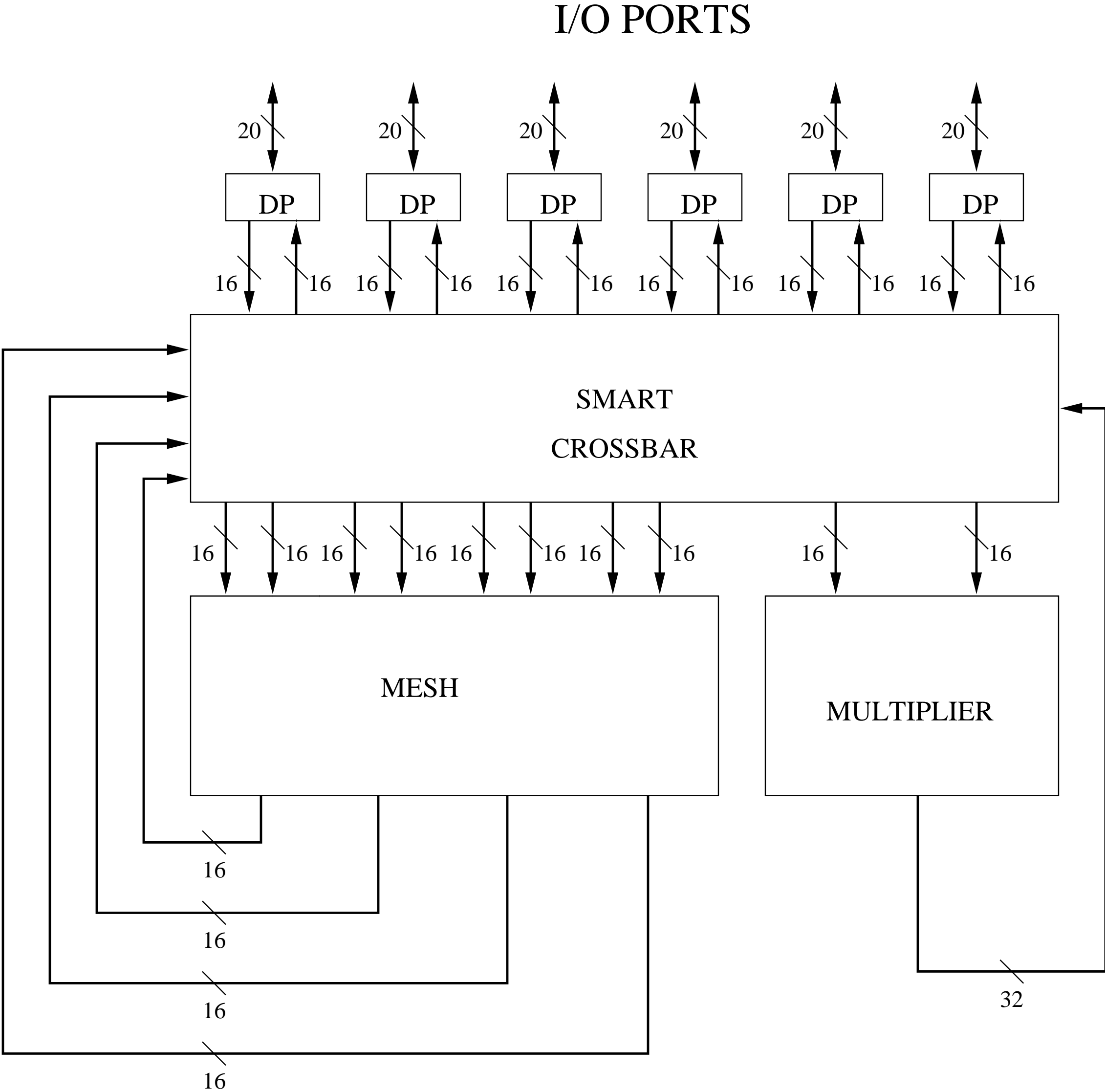


Figure 2.1: The Major Devices in the Colt Architecture.

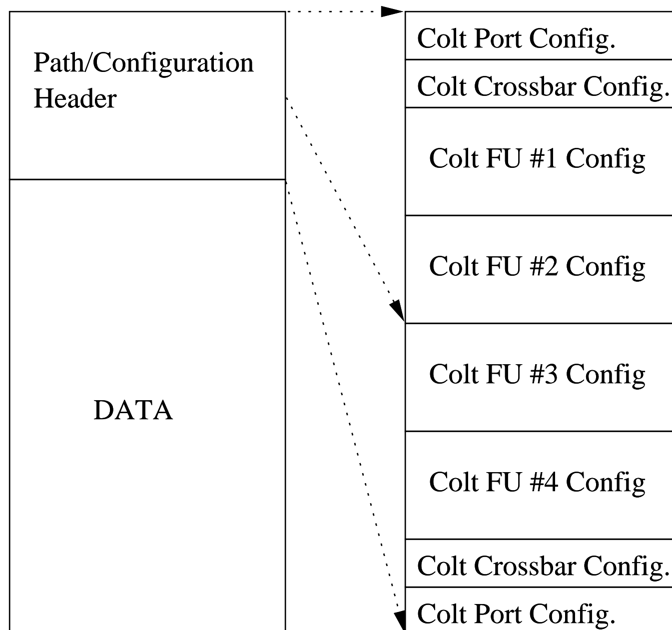


Figure 2.2: An Illustration of the Stream Format.

2.2 Crossbar and Data Port

There are six bi-directional data ports which control the flow of data between the inside and the outside of the Colt chip. Each is 20-bit wide, consisting of 16 bits for data and four bits for stream flow control [Mus96].

The on-chip crossbar is an intelligent network with 12 inputs and 16 outputs that support 16-bit data paths. Six of the inputs arrive from the data ports previously mentioned, four from the bottom of the computational mesh, and the remaining two come from the 32-bit output of the multiplier. As for the outputs from the crossbar, six are routed to the data ports, while the mesh uses eight and the multiplier uses two.

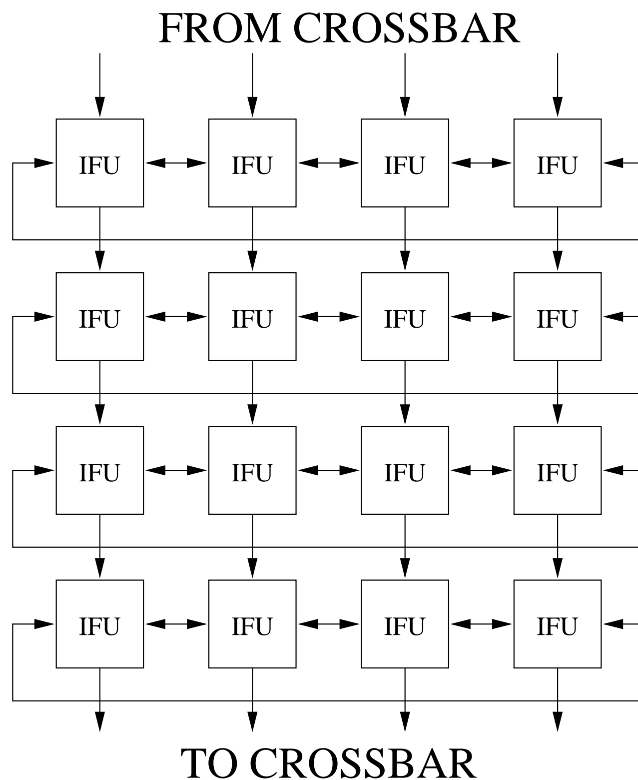


Figure 2.3: The Mesh Topology within the Colt Chip.

The crossbar provides full connectivity from any input to any output, except between the data ports. A direct connection from one data port to another is not supported since it was deemed to be a waste of chip resources, since no computation would be performed on this path [Bit97a].

The inputs and the outputs of the multiplier are connected to the crossbar. Because of the high degree of interconnectivity, it is easy to map algorithms under this architecture.

2.3 Mesh

The computing core of the Colt is an array of Interconnected Functional Units (IFUs) arranged in a mesh [Hwa93]. Figure 2.3 shows the structure of the mesh, which is a 4×4 array. Each node

of the mesh is an IFU that consists of routing buses and a Function Unit(FU).

Each IFU has a north, south, east, and a west local connection. All of the connections are bi-directional, except for the first row of the mesh, which only gets inputs from the crossbar. Likewise the last row is not bi-directional it only has output connections to the crossbar [Mus96].

Skip bus is another powerful connection scheme to enhance the routing of the mesh. Skip bus provides the ability to bypass the IFU without interfering with the function of the IFU. Because skip bus does not enter the IFU, it is not latched by the register. The signal transmitted by skip bus can be treated as that transmitted by a normal bus without any latency.

2.4 Interconnected Function Unit

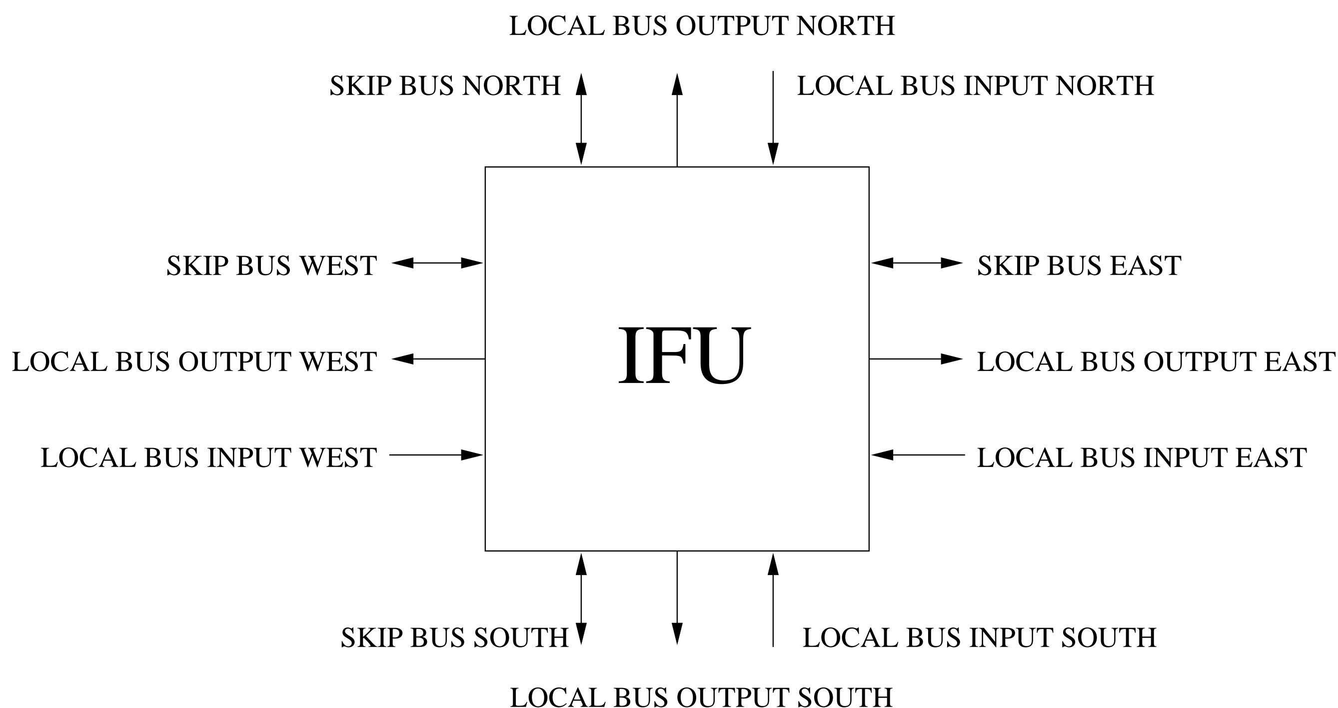


Figure 2.4: Interconnected Functional Unit.

The IFU provides the reconfigurable interconnection in the mesh. It consists of the FU surrounded by the control circuitry and buses necessary to implement the local and skip connections. Figure 2.4 gives a symbolic view of the IFU's connectivity. Each side of the IFU has a bi-directional skip bus in addition to input and output local connections [Mus96].

2.5 Function Unit

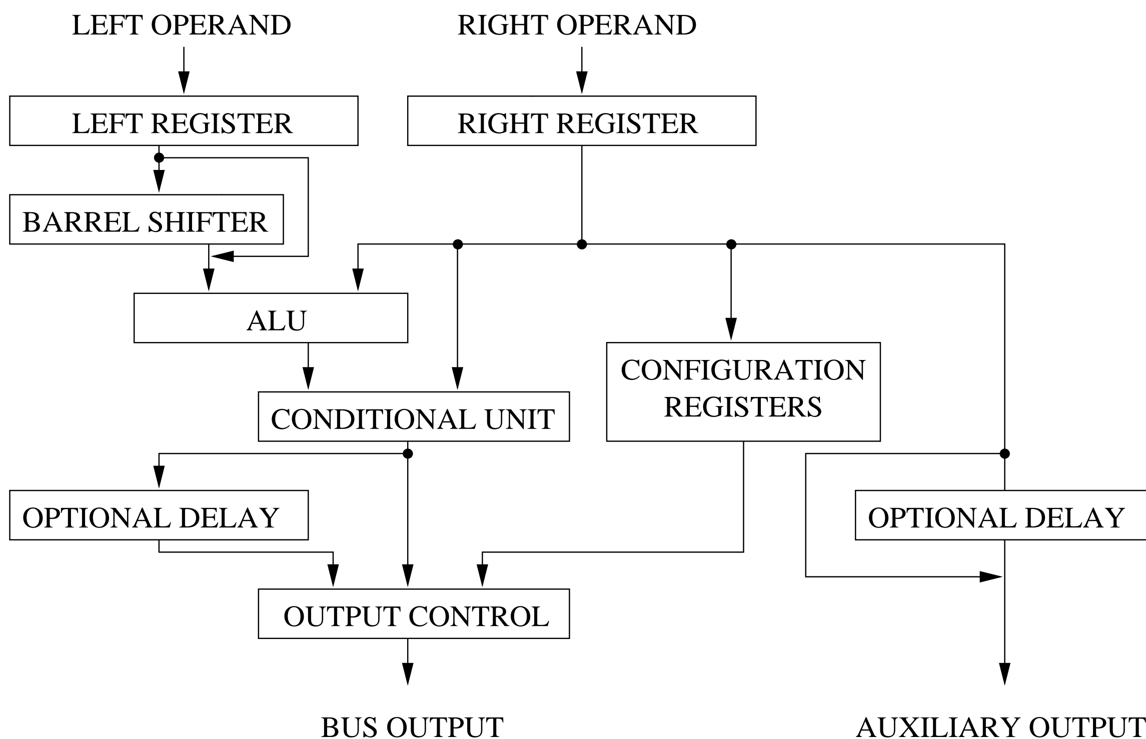


Figure 2.5: Functional Unit.

The Functional Unit (FU) is the basic computational unit used in the system. A partial schematic is shown in Figure 2.5. Two 16-bit operand inputs connect to the unit through the right and left operand registers. The left operand can be conditionally passed through a Barrel

Shifter. The (possibly) shifted left operand and the right operand are then sent through the arithmetic and logic unit (ALU). The results of the ALU operation then pass through a Conditional Unit, a Barrel Shifter and an optional Output Delay before leaving the FU [Bit97a].

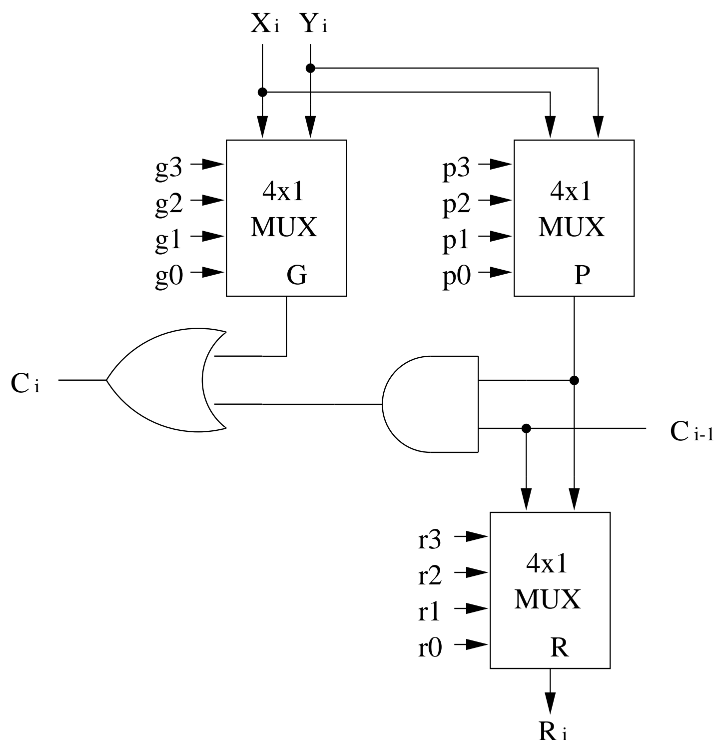


Figure 2.6: ALU Bit Slice.

The ALU is a highly configurable unit which consists of three data inputs, three program inputs and two data outputs. Figure 2.6 shows the ALU bit slice where C_{i-1} is the carry from the bit slice of the lower bit and C_i is sent to that of the higher bit. The outputs of the multiplexer can be configured by the values of the three program inputs, P, G and R. This schematic allows the ALU to be programmed for a variety of functions including addition, subtraction, pass-through, increment, decrement, and all boolean functions.

2.6 Multiplier

Table 2.1: Comparison of 16-bit Unsigned FPGA Multiplier Implementations.

Device	Serial		Parallel	
	Resources	Rate(MHz)	Resources	Rate(MHz)
Altera 81188	7%	2.14	51%	3.66
NS CLAy	3%	0.91	45%	3.60
Xilinx 4010	8%	1.90	60%	3.80

Multiplication is a common operation in DSP-type applications, so much so that almost all DSP processors contain a dedicated Multiply-and-Accumulate (MAC). Thus it is a safe assumption that the Colt chip will frequently be required to do a great deal of multiplication as well. Unfortunately, multiplication tends to be too resource intensive to implement in FPGA-type architectures and it is relatively slow. Table 2.1 shows some of the results compiled by Petersen and Hutchings [Pet95] on the implementation of multipliers in FPGAs [Bit97a].

To handle multiplications and save resources, it was decided by the chief architect, Ray Bittner, that a dedicated multiplier should be included in the chip architecture. This decision provides many advantages which conventional custom computing machine is hard to achieve. First, the reconfigurable resources of Colt can be focused on the data flow instead of being sacrificed to the resource-consuming multiplication. Second, the multiplication time is significantly improved by the VLSI implementation. Third, the algorithm is easily mapped by connecting the multiplier to the crossbar. The output of the multiplier can be directed to the functional unit to be processed further. Besides the word-oriented approach gives Colt a better opportunity to integrate macrocells than conventional bit-oriented FPGAs. Macrocells, which are hard to be implemented by IFUs such

as multipliers, dividers and floating-point units, can be easily embedded by connecting them to the crossbar. Although heterogeneous units reside in the chip, they can still be configured by the concept of wormhole RTR. The crossbar provides the ability to embed even more than one macrocells if needed.

It is difficult for conventional FPGAs to embed such macrocells in its architecture, and even then it is still difficult to utilize resources efficiently under bit-oriented architecture. Conventional FPGAs only consist of homogeneous cells. These cells are uniformly distributed all over the chip. Even though the multiplier can be placed in the FPGA by removing some cells, the problem then becomes how the multiplier is connected to the rest of the circuit. What cells should the multiplier be connected to? In what way? The implementation of CAD tools is another problem. The regularity of the cell array of FPGAs is broken by the multiplier. The problem of routing and placing is hard to be solved then.

The Colt multiplier has two inputs from the crossbar which are stored in the registers of Function Units. The multiplier has to pass all programming information from the inputs to the output. The multiplicand and the multiplier are both 16-bit wide integers. This will produce a 32-bit wide product which is stored in the output register of the multiplier. To fit the data stream of the Colt chip, the product has to be divided into two 16-bit wide output streams and sent to the crossbar.

To meet the processing needs of the DSP, the multiplier should be capable of performing a multiplication every clock cycle. The clock rate of Colt is 50 MHz and the clock cycle is 20 nanoseconds. A pipelined latency on the output is acceptable.

Chapter 3

Architecture of Multiplier

3.1 Overview

Table 3.1: 4-bit Multiplier Partial Products

				X_3	X_2	X_1	X_0	Multiplicand
				Y_3	Y_2	Y_1	Y_0	Multiplicand
				X_3Y_0	X_2Y_0	X_1Y_0	X_0Y_0	
			X_3Y_1	X_2Y_1	X_1Y_1	X_0Y_1		
		X_3Y_2	X_2Y_2	X_1Y_2	X_0Y_2			
	X_3Y_3	X_2Y_3	X_1Y_3	X_0Y_3				
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	Product

Table 3.1 depicts the 4×4 parallelogram for multiplication of integers represented in any positional number system. The multiplication process may be viewed to consist of the following two steps [Wes93]:

1. Evaluation of partial products.

2. Accumulation of the shifted partial products.

The product of two n-digit numbers can be accommodated in $2n$ digits. In the binary system, an AND gate can be used to generate partial product X_iY_j [Ham90]. The evaluation of partial

products consists of the logical ANDing of the multiplicand and the relevant multiplier bit. Each column of partial products must then be added and , if necessary, values passed to the next column.

In general, the multiplier can be divided into three categories:

1. Serial form.

2. Serial/parallel form.

3. Parallel form.

Since the multiplication has to be done in a clock cycle for Colt, it is impossible to achieve this goal by either serial form or serial/parallel form; thus the multiplier of parallel form is the only candidate for this research.

The speed of the multiplier is determined by both architecture and circuit. The speed can be expressed by the number of the cell delays along the critical path on the architecture level of the multiplier. The cell delay, which is normally a delay of an adder, is determined by the design of the circuit of the cell.

3.2 Multiplier Architectures

Three general categories of multiplier architectures: tree multiplier, Booth recoded multiplier, and array multiplier, will be discussed in this section. The advantages and disadvantages of these architectures have to be explored to clarify the choice of the architectures.

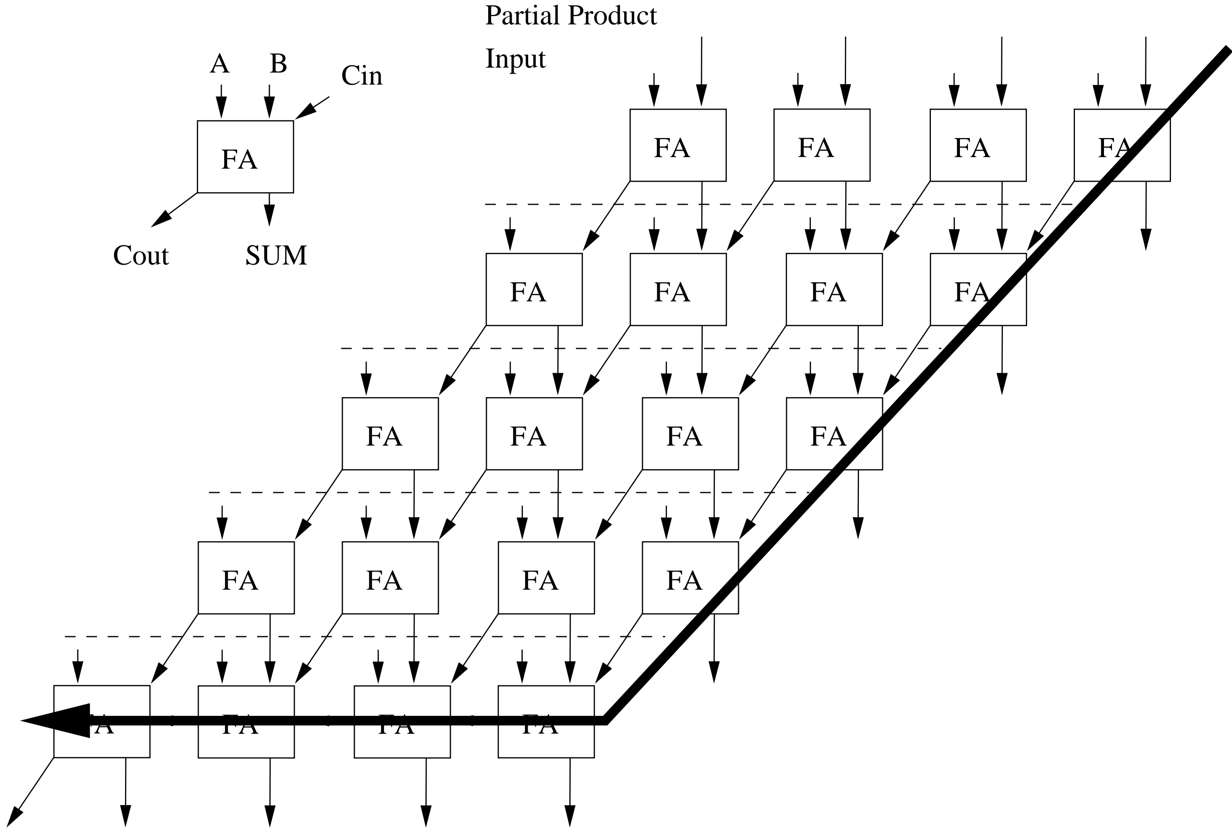


Figure 3.1: A 4x4 Array Multiplier.

3.2.1 Array Multiplier

The array multiplier originates from the multiplication parallelogram. As shown in Figure 3.1, each stage of the parallel adders should receive some partial product inputs. The carry-out is propagated into the next row. The bold line is the critical path of the multiplier. In a non-pipelined array multiplier, all of the partial products are generated at the same time. It is observed that the critical path consists of two parts: vertical and horizontal. Both have the same delay in terms of full adder delays and gate delays. For an n-bit by n-bit array multiplier, the vertical and the horizontal delay are both the same as the delay of an n-bit full adder. For a 16-bit by 16-bit multiplier, the

worst-case delay is 32τ where τ is the worst-case full adder delay.

One advantage of the array multiplier comes from its regular structure. Since it is regular, it is easy to lay out and has a small size. The design time of an array multiplier is much less than that of a tree multiplier. A second advantage of the array multiplier is its ease of design for a pipelined architecture. A fully pipelined array of the multiplier with a stage delay equal to the delay of a 1-bit full adder plus a register has been successfully designed for high-speed DSP applications at Bell Laboratories [Hat86].

The main disadvantage of the array multiplier is the worst-case delay of the multiplier proportional to the width of the multiplier. The speed will be slow for a very wide multiplier. For example, the worst-case delay of a 54-bit by 54-bit multiplier employing the array scheme will be over 100τ.

3.2.2 Tree Multiplier

Table 3.2: The Truth Table of a Full Adder

Number of Ones	Carry	Sum	A	B	C
0	0	0	0	0	0
1	0	1	0	0	1
1	0	1	0	1	0
2	1	0	0	1	1
1	0	1	1	0	0
2	1	0	1	0	1
2	1	0	1	1	0
3	1	1	1	1	1

The result of the multiplication is obtained by first generating partial products and then adding

the partial products. The critical path of a multiplier depends on the delay of the carry chain through all of the adders. Table 3.2 shows the truth table of a full adder, which is the basic addition process usually employed in a computer to add two numbers together. A and B are the adder inputs, and C is the carry input. The full adder produces a bit of summand and a bit of carry out. It can be observed that a full adder is actually a "one's counter". A, B and C can all be seen as the inputs of 3-2 compressor. The outputs, Carry and Sum, are the encoded output of the three inputs in binary notation [Wes93]. The tree multiplier is based on this property of the full adder. The addition of summands can be accelerated by adopting a 3-2 compressor.

A 3-2 compressor adds three bits and produces a two-bit binary number whose value is equal to that of the original three. The advantage of the 3-2 compressor is that it can operate without carry propagation along its digital stages and hence is much faster than the conventional adder. In any scheme employing 3-2 compressors, the number of adder passes occurring in a multiplication before the product is reduced to the sum of two numbers, will be two less than the number of summands, since each pass through an adder converts three numbers to two, reducing the count of numbers by one. To improve the speed of the multiplication, one must arrange many of these passes to occur simultaneously by providing several 3-2 compressors.

Thus, the best first step for a tree multiplier is to group the summands into threes, and introduce each group into its own 3-2 compressor, thus reducing the count of numbers by a factor of 1.5. The second step is to group the numbers resulting from the first step into threes and again add each group in its own 3-2 compressors. By continuing such steps until only two numbers remain, the addition is completed in a time proportional to the logarithm of the number of summands [Wal64].

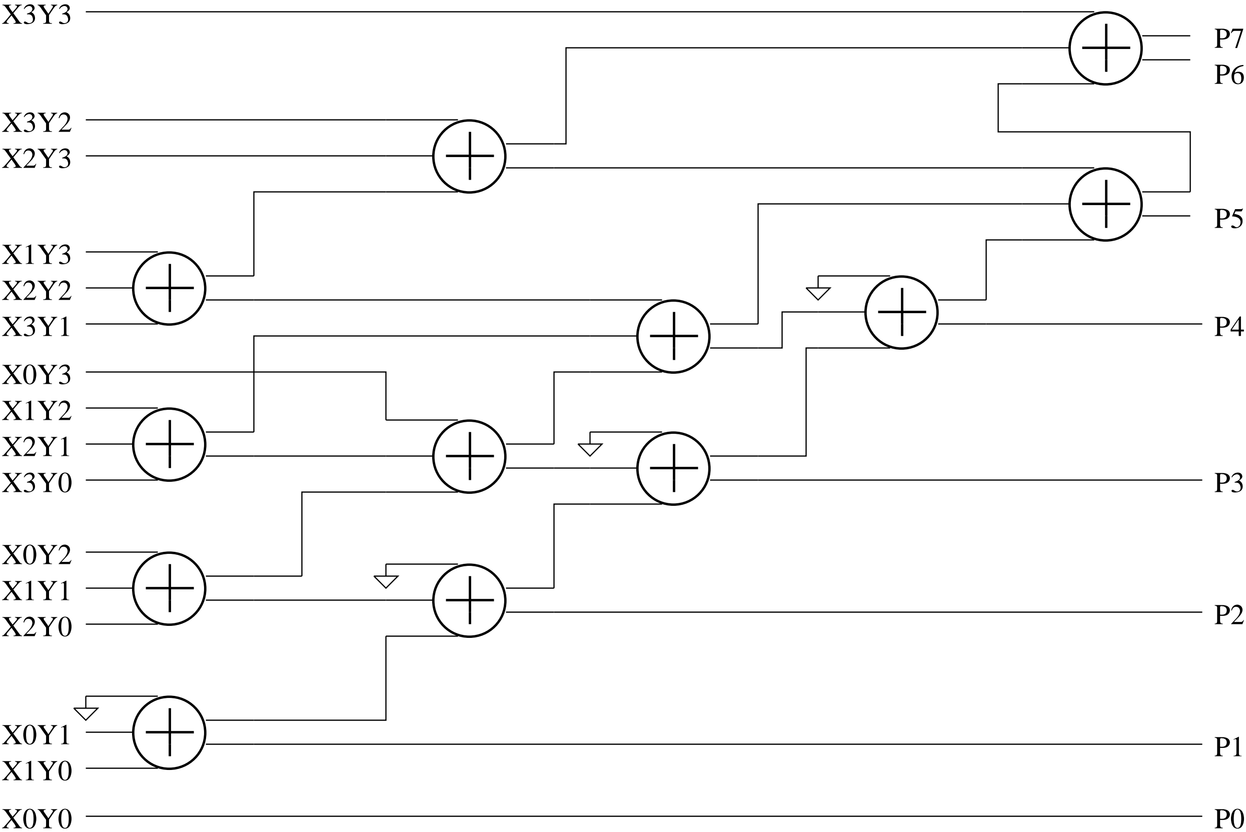


Figure 3.2: A 4x4 Tree Multiplier

Figure 3.2 shows a 4-bit by 4-bit tree multiplier. It should be noted that 4:2 compression can also be used besides the 3:2 compression. A 4:2 compression can be achieved by combining two full adders. For a 16-bit by 16-bit multiplier, the worst-case delay is $7\tau_{fa} + \tau_{cpa}$ where τ_{fa} is the delay of a 3-2 compressor, which is the delay of a full adder, and τ_{cpa} is the delay of the fast carry propagate adder.

Although the tree multiplier has great improvement of speed. The disadvantage of the tree multiplier is the irregular structure as shown on Figure 3.2 which leads to an irregular layout,

although many efforts have been spent on regularizing the layout in the past [Got92], [Vui82]. Hence it is not possible to create modular layout. The design time of a tree multiplier is much more than that of a array multiplier. The routing of the signals will be a time-consuming process.

3.2.3 Booth Recoded Multiplier

The motivation of the Booth encoder is that in cases where the multiplier has its 1*s* grouped into a few contiguous blocks, only a few versions of the multiplicand need to be added to generate the product. If only a few summands need to be added, the multiplication operation can be speed up [Ham90].

To multiply two numbers m and r together. examine the nth digit (m_n) of m [Boo51],

1. If $m_n = 0, m_{n+1} = 0$, multiply the existing sum of partial products by 2^{-1}, i.e. shift one place to the right.

2. If $m_n = 0, m_{n+1} = 1$, add r into the existing sum of partial products and multiply by 2^{-1}, i.e. shift one place to the right.

3. If $m_n = 1, m_{n+1} = 0$, subtract r from existing sum of partial products and multiply by 2^{-1}, i. e. shift one place to the right.

4. If $m_n = 1, m_{n+1} = 1$, multiply the sum of partial products by 2^{-1}, i.e. shift one place to the right.

5. Do not multiply 2^{-1} at m_0 in the above processes.

Table 3.3: Booth Multiplier Recoding Table [Ham90]

Bit i	Bit $i-1$	Version of Multiplicand selected by bit i
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

Table 3.4: The Multiplier and The Recoded Multiplier by Booth Algorithm[Ham90]

The Multiplier	1	1	0	0	0	1	0	1	1	0	1	1	1	1	0	0
The Recoded Multiplier	0	-1	0	0	+1	-1	+1	0	-1	+1	0	0	0	-1	0	0
Bit-pair recoding		0		-1		0		-1		+2		+1		0		-1

Table 3.5: Booth Multiplier Bit-Pair Recoding Table [Ham90]

Bit $i+1$	Bit i	Bit $i-1$	Version of Multiplicand selected by bit i
0	0	0	$0 \times M$
0	0	1	$+1 \times M$
0	1	0	$+1 \times M$
0	1	1	$+2 \times M$
1	0	0	$-2 \times M$
1	0	1	$-1 \times M$
1	1	0	$-1 \times M$
1	1	1	$0 \times M$

Table 3.3 shows the Booth technique for recoding multipliers. Table 3.4 shows an example of Booth recoded algorithm. This technique can be applied to more than two bits simultaneously. Grouping the Booth-recoded selector in pairs (Table 3.5), we obtain a single, appropriately shifted summand for each pair as shown. The basic idea of the speedup technique is to use bits $i+1$ and i to select one summand, as a function of bit $i-1$, to be added at summand position i [Ham90]. For a n-bit multiplier, the pair-recoding will generate at most $n/2$ summands. This is twice as fast as the worst-case original Booth algorithm situation.

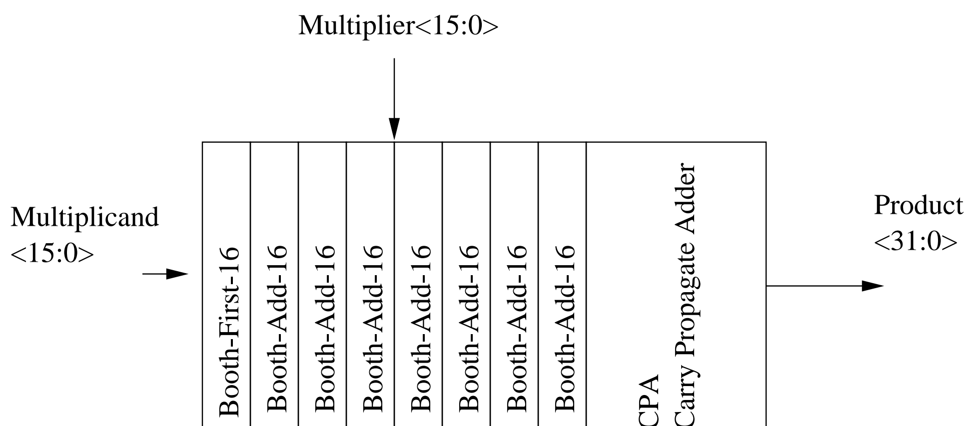


Figure 3.3: A 16x16 Multiplier Top Level Schematic and Floor Plan

The advantage of the Booth recoded multiplier is that it avoids the need to correct the product when either input is negative. As shown on Figure 3.3, the floor plan can be divided into two layers: a Booth array and a CPA. Both layers have the property of regularity.

3.3 Linear Pipeline

Pipeline techniques can be applied to the design of the multiplication [Hat86], [San89]. A speedup can be easily achieved by implementing the pipeline. A linear pipeline processor is a cascade of processing stages which are linearly connected to perform a fixed function over a stream of data flowing from one end to the other [Hwa93].

3.3.1 Synchronous Model

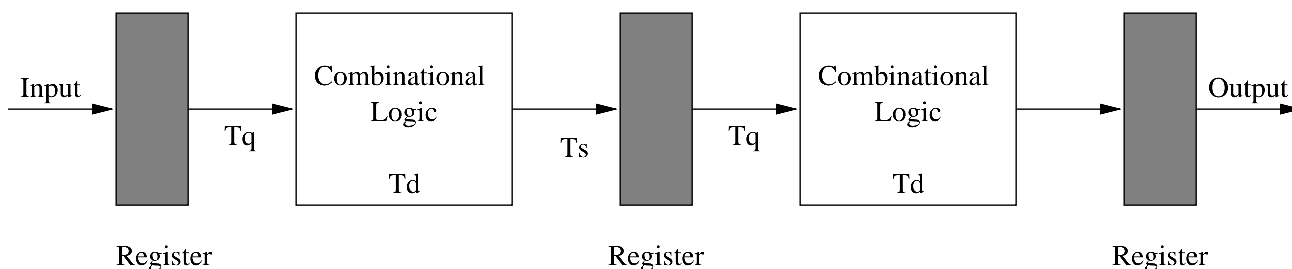


Figure 3.4: A Synchronous Pipeline Model.

Synchronous pipelines are illustrated in Figure 3.4. Clocked registers are used to interface between stages. Upon the arrival of a clock pulse, all registers transfer data to the next stage simultaneously.

Successive tasks or operations are initiated one per cycle to enter the pipeline. Once the pipeline is filled, one result emerges from the pipeline for each additional cycle. This throughput is sustained only if the successive tasks are independent of each other [Hwa93].

3.3.2 Clocking and Timing Control

The hold time relates to the delay between the clock input to the register and the storage element. The data has to be held for this period while the clock travels to the point of storage. The setup time is the delay between the data input of the register and the storage element. As the data takes a finite time to travel to the storage point, the clock can not be changed until the correct data value appears. The delay from the positive clock input to the new value of the register output is called the clock-to-Q delay [Wes93].

The clock cycle τ of a pipeline is determined by the equations below. Let τ_i be the time delay of the circuitry in stage S_i, T_s the setup time of the register, and T_q the Clock-to-Q delay. Denote the maximum stage delay as τ_m, then

$$\tau_m = \max_i (\tau_i)_1^k \tag{3.1}$$

$$\tau = \tau_m + T_s + T_q \tag{3.2}$$

The pipeline frequency is defined as the inverse of the clock period:

$$f = \frac{1}{\tau} \tag{3.3}$$

3.3.3 Speedup

Ideally, a linear pipeline of k stages can process n tasks in $k + (n-1)$ clock cycles, where k cycles are needed to complete the execution of the very first task and the remaining $n-1$ tasks require $n-1$ cycles [Hwa93]. Thus the total time is

$$T_k = [k + (n-1)]\tau \tag{3.4}$$

where τ is the clock period. Consider an equivalent-function non-pipelined processor which has a flow-through delay of $k\tau$. The amount of time it takes to execute n tasks on this non-pipelined processor is $T_1 = nk\tau$.

The speedup factor of a k-stage pipeline over an equivalent non-pipelined processor is defined as

$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{k\tau + (n-1)\tau} = \frac{nk}{k + (n-1)} \tag{3.5}$$

If n is very large, then $S_k = k$ which is the number of the pipeline stages.

3.3.4 Pipelined Multiplier

Pipelined multipliers are useful in systems where arithmetic throughput is more important than latency. Colt is a good example of the systems to implement a pipelined multiplier. As discussed in this section, a pipelined multiplier can perform more operations in a certain amount of time and operate with a higher clock rate than a nonpipelined multiplier.

The increased throughput does not come without price: the registers used to pipeline the multiplier significantly increase the silicon area taken up by the multiplier. If there are too many pipeline stages, the clock rate of the multiplier will be higher than necessary and the multiplier will take up more area than required. On the other hand, if there are too few pipeline stages, the multiplier will not be able to operate at the system clock rate.

To cope with the trade-off, the designer must have a detailed knowledge of the design of the nonpipilined multiplier so that they will know how the pipelining should be done and how fast the resulting circuit will operate. If their knowledge of the circuit is not sufficient, the resulting circuit

may not function as expected or will not be optimized for their application [Asa90].

3.4 Solution for the Architecture

The first concept behind the choice of the architecture is the trade-off between area and speed. An optimal system should occupy minimum area but still operate under speed specification. From the discussion of the multiplier architecture (Section 3.2), the smallest architecture of the multiplier is array multiplier. However, it is also the slowest architecture.

The second concept behind the choice of the architecture is the regularity, which means the design can be divided into several identical submodules. By iterating these submodules, a system can be easily constructed. The design time is significantly reduced in this way. Extended use may be made of regular structures to simplify the design process. Regularity allows an improvement in productivity by reusing specific designs in a number of places, thereby reducing the number of different designs that need to be completed. Array multipliers and Booth recoded multipliers both satisfy the property of the regularity.

A pipelined array multiplier can meet all of these requirements. It has the advantages of array multipliers such as regular implementation and minimum area. By inserting one or two pipeline stages into the multiplier, the speed can be significantly improved. The problem of the array multiplier is solved then. The flip-flop used in Colt only needs six transistors. This flip-flip reduces the required area of inserting extra pipeline stages. The time of the multiplication should be less than $(\tau - T_s - T_Q)$, where τ is the clock cycle, T_s is the setup time of the output register and T_Q is the clock-to-Q time of the input register. The number of pipeline stages is determined by the

speed of the cell. The detail of the circuit design will be discussed on the next chapter.

Chapter 4

Circuit Design

Circuit design plays an important role in the design of the multiplier. First, to guarantee the multiplier to work at the desired clock rate, the designer has to know the delay of the critical path and the required time of inserting a pipeline stage. Second, to reduce the area of the multiplier, several architectures of adders are investigated. Circuit analysis helps the designer verify the functions and performances of the adders. The architecture of the adder has to be determined first. Then the number of the pipeline stages can be decided by the speed of the adder. The size of the multiplier should be as small as possible if all the requirements can be met.

4.1 CMOS Logic Structures

The Colt chip is fabricated with the HP 0.5 μm CMOS process, which is a triple-metal single-poly process (Appendix A). There are a number of alternative CMOS logic structures under this process. CMOS complementary logic structure, pass-transistor logic structure and complementary pass-transistor logic structure are used in the design of the multiplier.

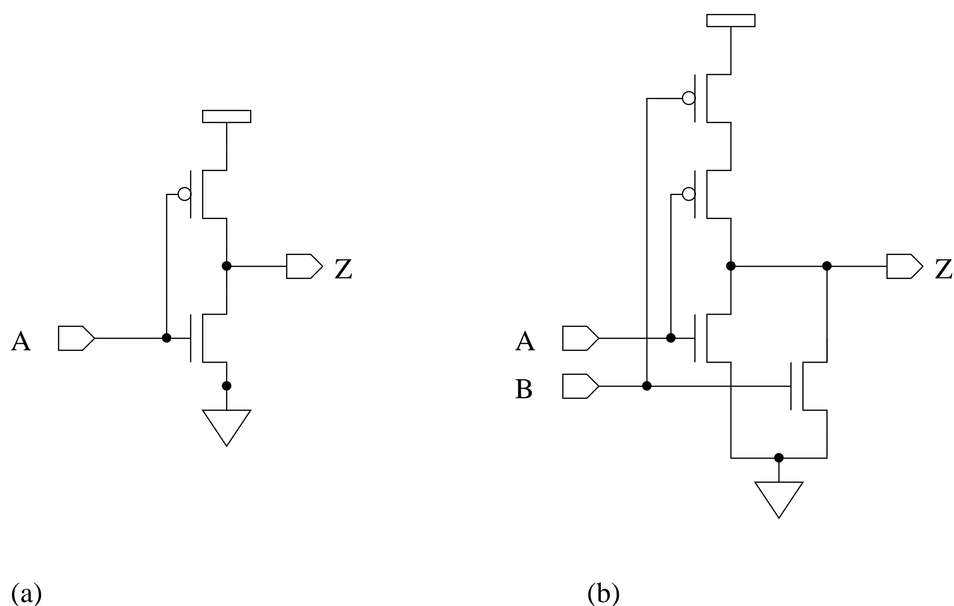


Figure 4.1: (a) CMOS Inverter. (b) CMOS NOR Gate.

4.1.1 CMOS Complementary Logic

Figure 4.1 (a) shows a CMOS inverter. There is a pullup PMOS transistor and a pulldown NMOS transistor. The steady state of the output will be independent of the ratio of the pullup and pulldown transistor sizes [Gei90]. Because of this, CMOS complementary logic does not have to worry about signal degradation problems in pass-transistor logic. Because the power-to-ground path only closes during the transition, it almost consumes no static power. The CMOS complementary gate has two function determining blocks – an n-block and a p-block. There are normally $2n$ transistors in an n-input gate.

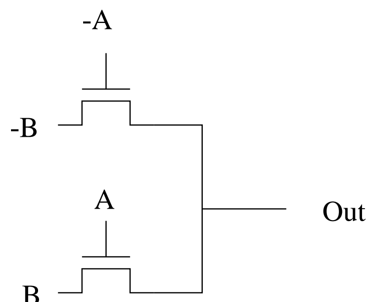


Figure 4.2: Two-input XNOR Gate Implemented in Pass-Transistor Logic.

4.1.2 Pass-Transistor Logic

Figure 4.2 shows an XNOR implemented in pass-transistor logic. A is used as a control signal and B is used as a pass signal. In this circuit, when A is true, B is passed to the output. When A is false, the complement of B is passed. As can be seen this logic form is not always active, which limits its ability to be used in long chains.

The advantage of pass-transistor logic is that it occupies much less area to construct complex Boolean function. The XNOR in CMOS complementary logic needs 3 2-input NAND gates, which consisted of 12 transistors. It only need 2 transistors in pass-transistor logic as shown in Figure 4.2.

4.1.3 Complementary Pass-Transistor Logic (CPL)

Several differential CMOS logic families such as cascade voltage switch logic (CVSL) [Hel84] (Figure 4.3) have been proposed for CMOS circuit speed improvement. CVSL is a differential style of logic requiring both true and complementary signals to be routed to gates [Wes93]. Two complementary NMOS switch structures are constructed and then connected to a pair of PMOS

Figure 4.3: CVSL Schematic Structure.

cross-coupled load, which together can reduce input capacitance, increase logic functionality, and sometimes eliminate inverter circuits. Therefore, these logic families can increase speed. However, the actual advantage of CVSL circuits is less than that anticipated in the original paper, as clarified in [Chu87]. This is because the PMOS cross-coupled latch can not easily be inverted due to the "fighting" to the NMOS pull-down trees. High-speed inversion of the PMOS latch is possible only when the gate width of the PMOS is sufficiently small. However, a small gate width severely degrades the pull-up transit time [Yan90].

The main concept behind CPL is the use of an NMOS pass-transistor network for logic organization, and elimination of the PMOS latch such as cascade voltage switch logic (CVSL) , as shown in Figure 4.4. CPL consists of complementary inputs/outputs, an NMOS pass transistor logic network, and CMOS output inverters. The pass transistors and the inverters function as

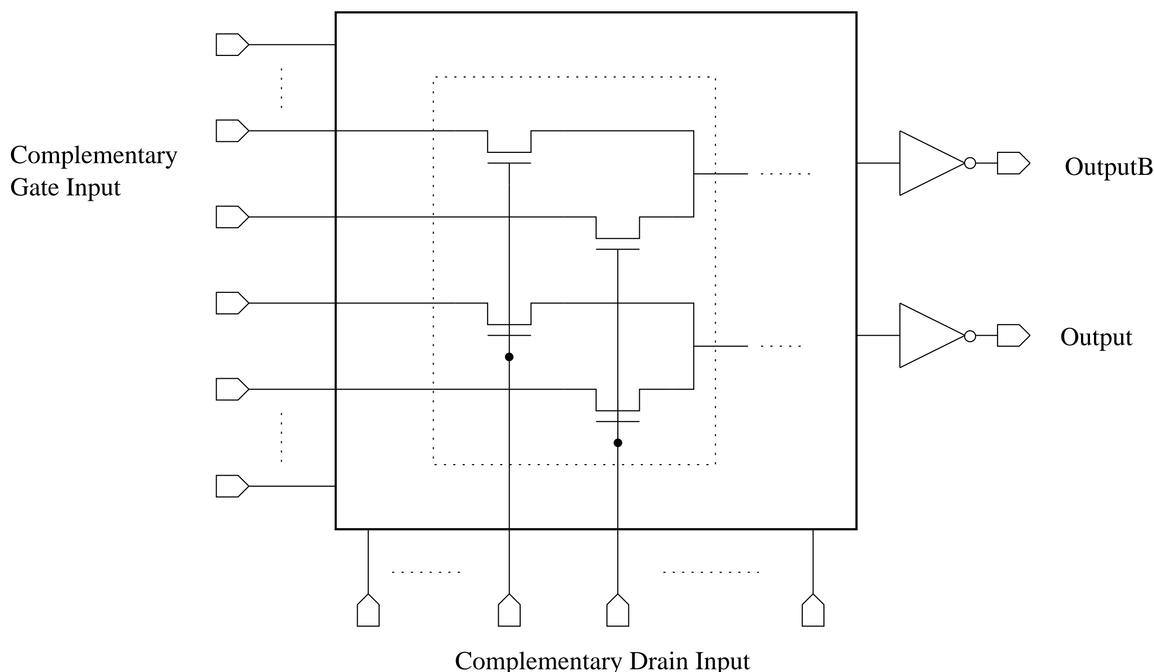


Figure 4.4: CPL Schematic Structure.

pull-down and pull-up devices. Thus the PMOS latch can be eliminated, allowing the advantage of the differential circuits to be fully utilized. Because the high level of the pass-transistor outputs (nodes Q and $\overline{Q}$) is lower than the supply voltage level by the threshold voltage of the pass transistors, the signal have to be amplified by the output inverters. At the same time, the CMOS output inverters shift the logic threshold voltage and drive the capacitive load [Yan90]. Figure 4.5 shows an AND/NAND circuit in CPL.

4.2 Adder

The adder is the basic element in computer arithmetic. It is also the critical element of the multiplier. The truth table of a one-bit full adder is shown on Table 4.1. We can derive the Boolean

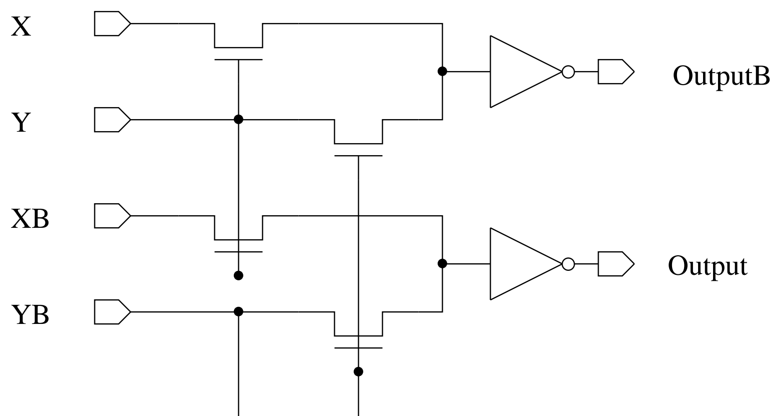


Figure 4.5: CPL AND/NAND Circuit.

Table 4.1: The Truth Table of a Full Adder [Wes93].

C	A	B	$A \times B(G)$	A+B(P)	$A \oplus B$	SUM	$CARRY$
0	0	0	0	0	0	0	0
0	0	1	0	1	1	1	0
0	1	0	0	1	1	1	0
0	1	1	1	1	0	0	1
1	0	0	0	0	1	1	1
1	0	1	0	1	1	0	1
1	1	0	0	1	1	0	1
1	1	1	1	1	0	1	1

function in sum of products from this truth table as the following:

$$SUM = ABC + A\overline{BC} + \overline{AB}C + \overline{A}B\overline{C} \tag{4.1}$$

$$CARRY = AB + BC + AC \tag{4.2}$$

A and B are the adder inputs, C is the carry input, SUM is the sum output, and $CARRY$ is the carry output. The generate signal, G, occurs when a carry output ($CARRY$) is internally generated within the adder. When the propagate signal, P, is true, the carry-in signal (C) is passed

to the carry output ($CARRY$) when C is true [Wes93]. Because the carry-in of the current bit is determined by lower bits of two operands, the delay of the adder depends on the generation of the carry.

4.2.1 Complementary CMOS Full Adder

Equation 4.2 may be factored as follows:

$$CARRY = AB + C(A + B) \tag{4.3}$$

$$\overline{CARRY} = \overline{AC} + \overline{BC} + \overline{AB} \tag{4.4}$$

From Equation 4.1 and Equation 4.4, SUM can be expressed as follows:

$$SUM = ABC + (A + B + C)\overline{CARRY} \tag{4.5}$$

This implementation is shown in Figure 4.6.

4.2.2 Transmission-Gate Adder

Figure 4.7 shows a transmission gate adder consists of 18 transistors. The inputs A and B are connected to an XNOR gate. The transmission gates and the inverters on the output form two XOR gates.

The use of this adder should be carefully studied. The outputs may not be active. The transmission gate between C and SUM may form a long chain of the transmission gates for a wide adder. The same conclusion can be applied to the transmission gate between C and CARRY. The long chain of the transmission gates will act as a RC circuit. Because the resistance and

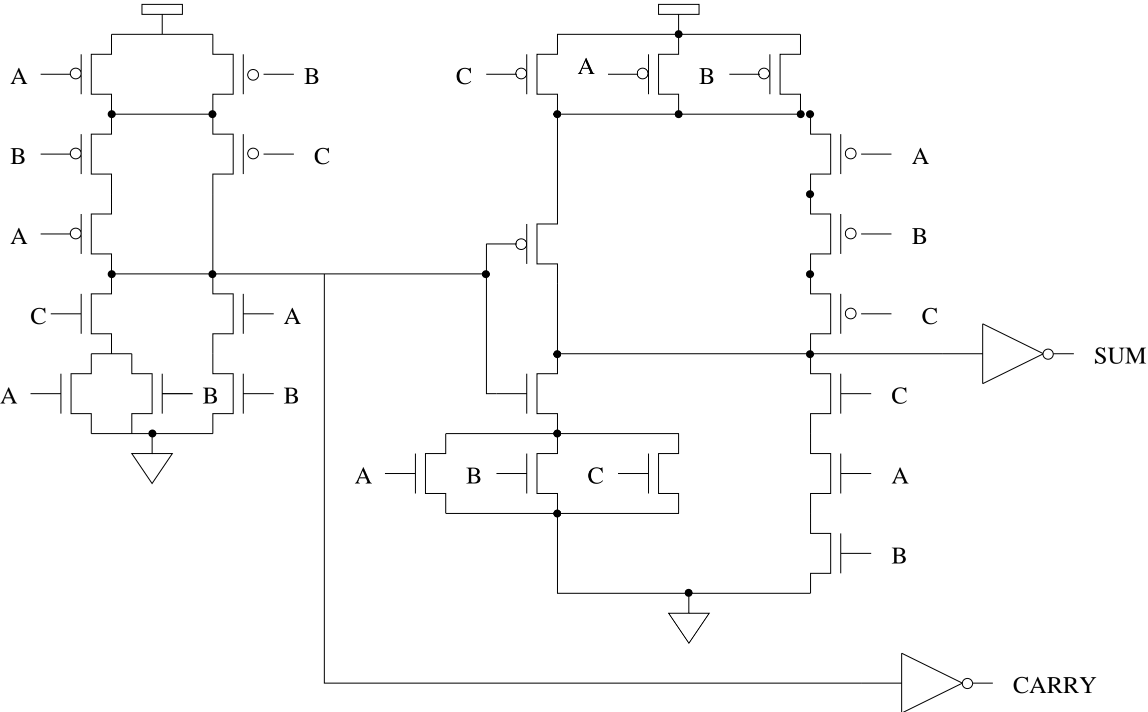


Figure 4.6: CMOS Full Adder[Wes93]

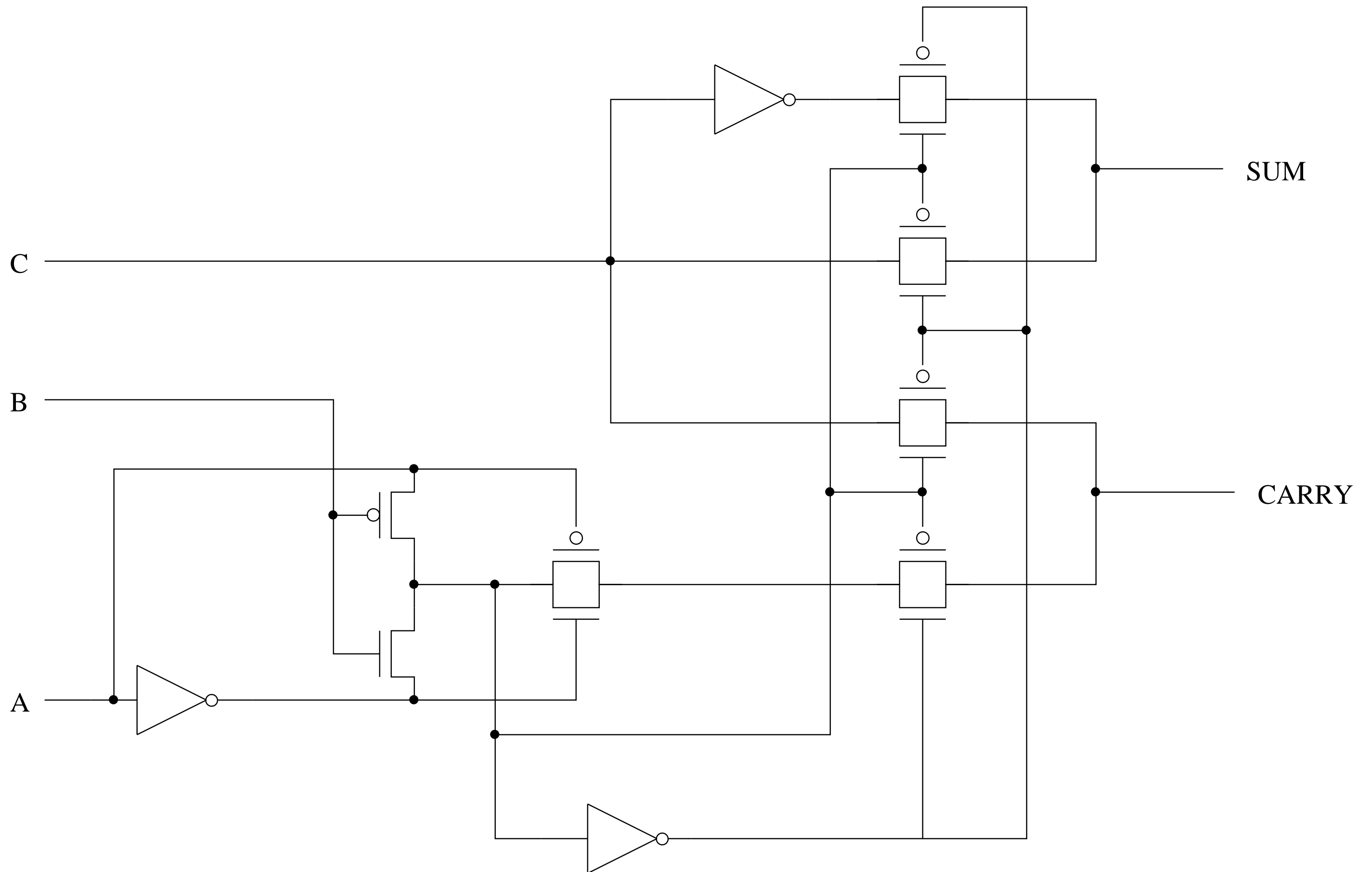


Figure 4.7: Transmission Gate Adder with 18 Transistors [Wes93]

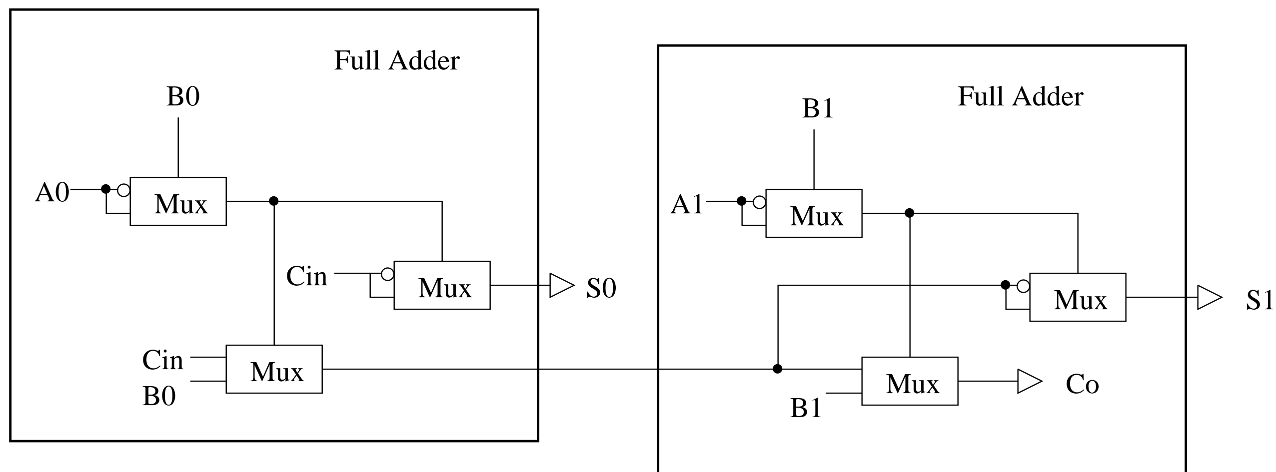


Figure 4.8: Full Adder Implemented by Transmission Gates

capacitance of the chain becomes large, the speed will slow down. It is necessary to insert some buffers in a wide adder.

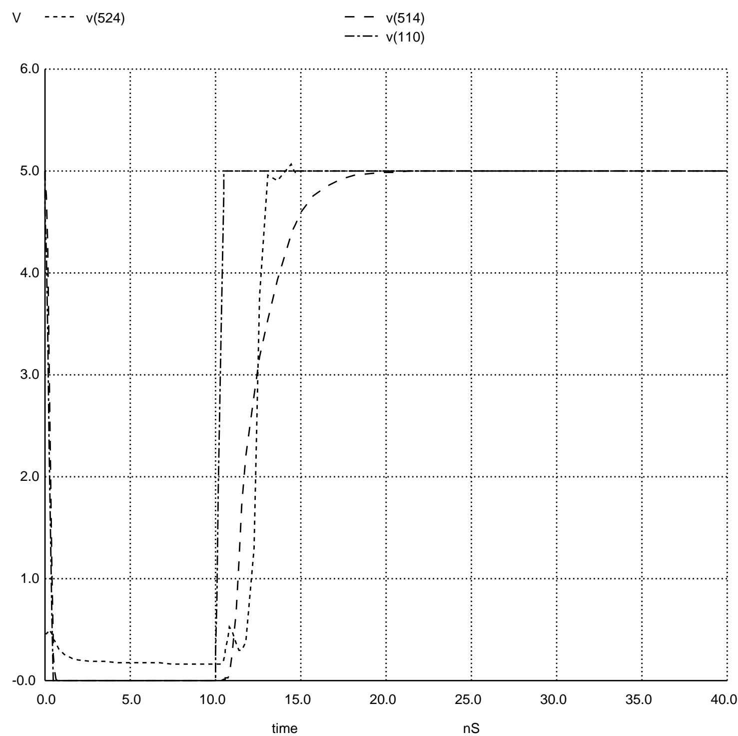
4.2.3 Modified CPL Full Adder

Figure 4.8 shows two full adders which are implemented in the CPL family. The difference from CPL is that the inputs of the multiplexers are either two different signals, or one signal and its complement. In CPL, all inputs of the multiplexer are complementary signals. However, each output still consists of the truth and the complement. This circuit is smaller than the CPL adder [Yan90]. All outputs (Co, S0, S1) have inverters to enhance driving ability. The multiplexer of this circuit can be implemented in two ways. One is implemented by CMOS transmission gates. The other is done by NMOS only. The NMOS pass-transistors occupies less area but has no equal rising time and falling time. It also suffers from signal degradation for logic 1. However, it has been shown to result in high speed due to its low input capacitance and high logic functionality

[Ohk95]. The signal degradation problem can be solved by the output inverter.

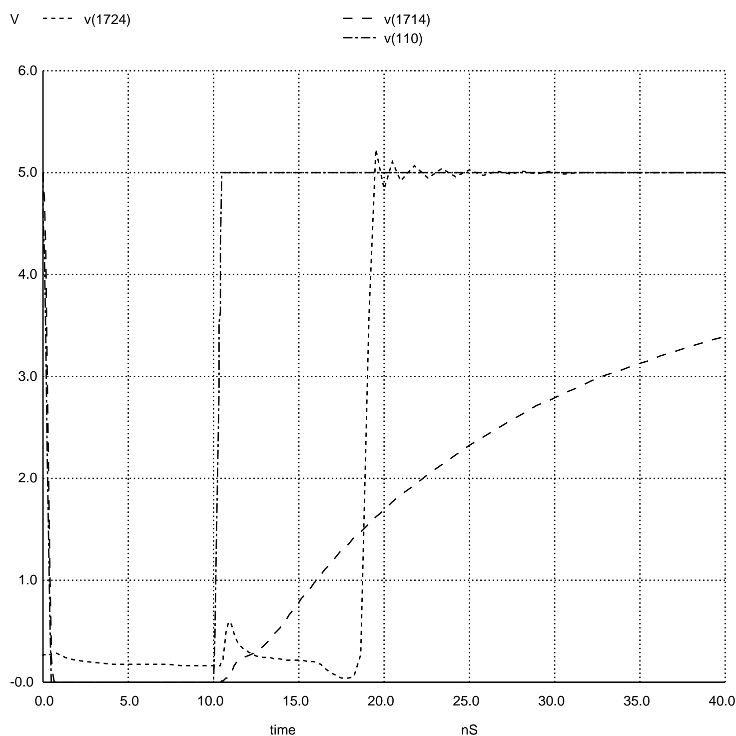
The purpose of using pass-transistor adders is to minimize the area. This circuit needs 36 transistors with the enhanced driving of each output. The transistor count per full adder is only 18. It is much smaller than the one in CMOS complementary logic. The full adder needs 28 transistors in CMOS complementary logic.

4.2.4 Comparison of the Adders



v(524): The output of 4-bit Modified CPL adder.
v(514): The output of 4-bit transmission-gate adder.
v(110): The input.

Figure 4.9: The Delay of 4-bit Modified CPL Adder and Transmission-Gate Adder.



v(1724): The output of 4-bit modified CPL adder.
v(1714): The output of 4-bit transmission-gate adder.
v(110): The input.

Figure 4.10: The Delay of 16-bit Modified CPL Adder and Transmission-Gate Adder.

The transmission-gate adder and the modified CPL adder are more attractive than the complementary CMOS logic adder if considering the required area, so the comparison of the adder is focused on the difference between the transmission-gate adder and the modified CPL adder.

The curve of the 4-bit transmission-gate adder climbs slower than that of the 4-bit modified CPL adder as shown Figure 4.9, because of the the effect discussed in Subsection 4.2.2. The RC effect of the transmission-gate adder is more distinctive for a 16-bit adder as shown 4.10. It is necessary to add buffers on the outputs of the transmission-gate adder. As a result, the transistor

count of the transmission-gate adder is 22.

4.3 Multiplexer

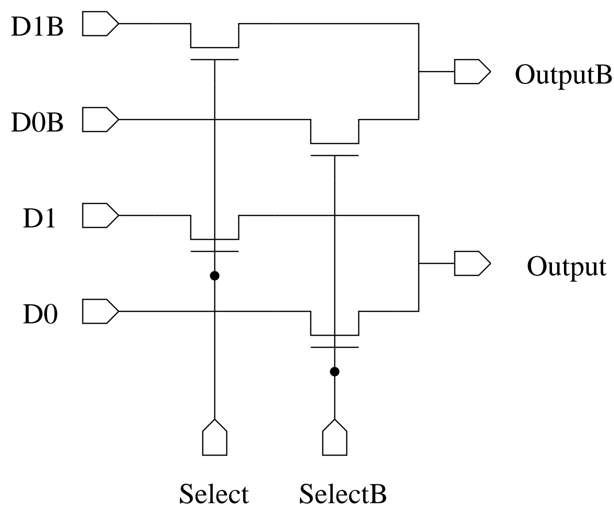


Figure 4.11: NMOS Multiplexer.

Figure 4.11 shows the schematic of the multiplexer used in the adder of Figure 4.8. The outputs of the multiplexer are the selected signal and its inverse. When Select is low, data D0 is selected, and when the control signal Select is high, data D1 is selected. The outputs are used as the inputs of the next-stage multiplexer. The multiplexer has both positive and negative outputs. It can reduce the propagation delay by eliminating an inverter.

4.4 Multiplier Cell

Figure 4.12 shows the block diagram of the multiplier cell. It consists of a two-bit adder and two AND gates. DAnd2 produces two signals. One is the same the output of an AND gate and the

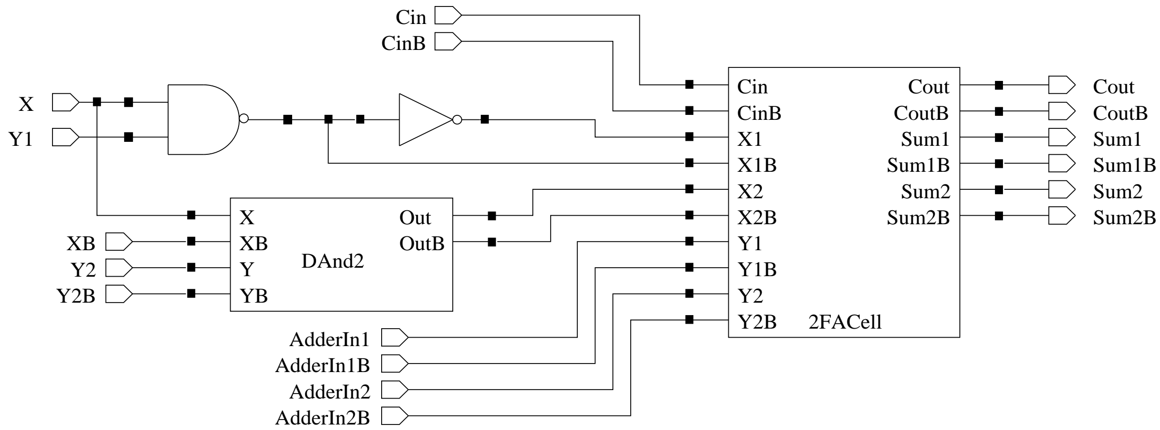


Figure 4.12: The Cell of the Multiplexer.

other is the same as the output of a NAND gate. The NAND gate and the inverter in this circuit provide strong signals entering 2FACell. This signal will be degraded for logic 1 within 2FACell.

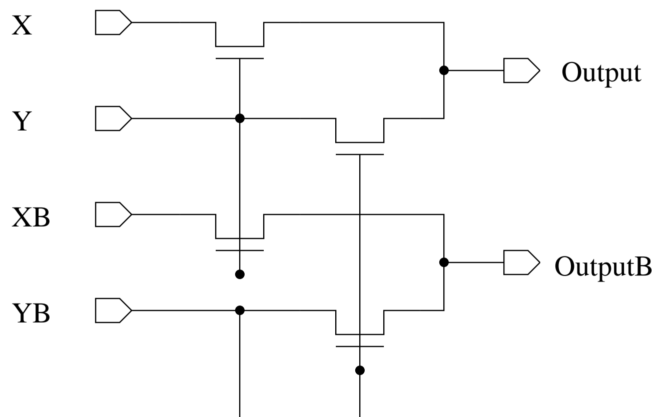


Figure 4.13: The AND Gate by Pass-Transistor Logic.

Figure 4.13 shows the schematic of the DAND which is actually a AND gate. This circuit only needs four transistors to generate the truth and complement of the output.

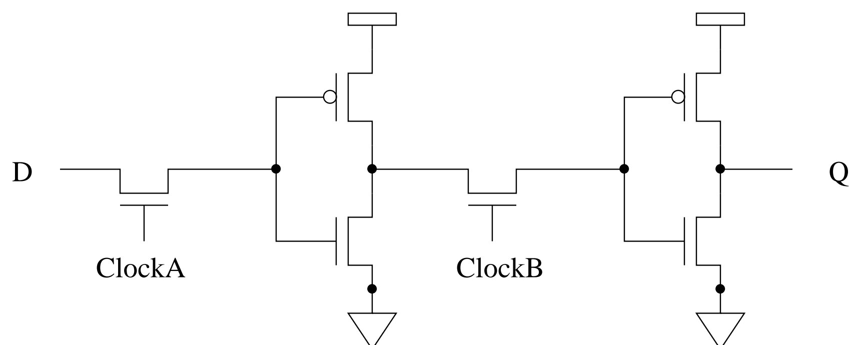


Figure 4.14: D Flip-Flop [Bit97a].

Table 4.2: Timing Property of D Flip-Flop [Che96].

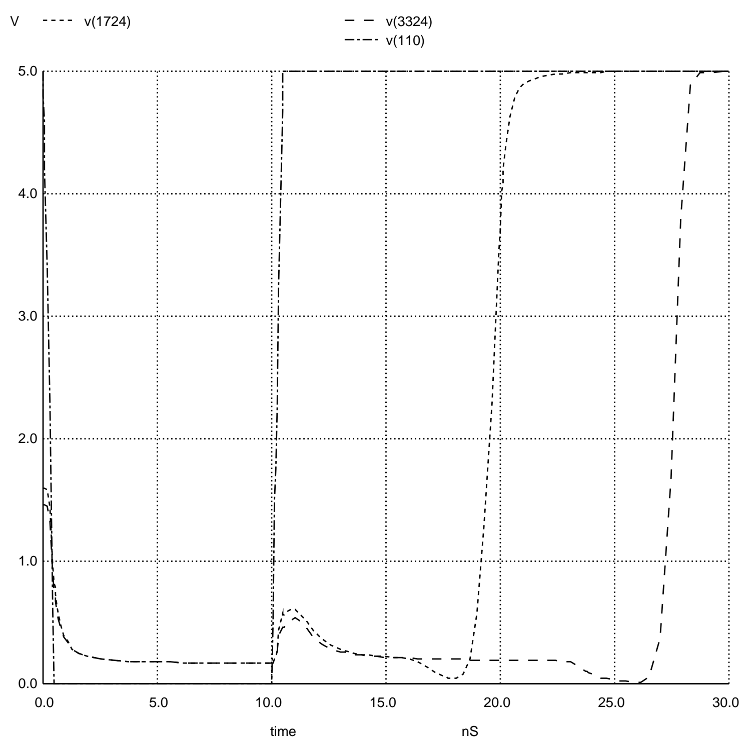
Property	Value
Clock-to-Q Time	2.5ns
Set-up Time	0.3ns
Hold Time	0.5ns
Cycle Time	3.3ns

4.5 Flip-Flop

The D flip-flop is shown in Figure 4.14. This is a dual phase dynamic flip flop, storing charge representing the current state at the inputs of the two inverters. There are two clock signals needed to drive the device, called ClockA and ClockB, which must be non-overlapping, two phase signals [Bit97a]. Table 4.2 shows the timing property of the D flip-flop. The sum of Clock-to-Q time and set-up time is 2.8 ns, so the time of the multiplication must be finished within 17.2 ns.

4.6 Architecture

The design of the multiplier can be finalized based on the previous discussion. The modified CPL adder is the smallest and fastest among all three adders. Figure 4.15 shows the delays of the 16-bit and 32-bit modified CPL adder. The delay of the 16-bit modified CPL adder is about 9 nanoseconds, and the delay of the 32-bit modified CPL adder is about 17.5 nanoseconds. If the



v(3324): The output of the 32-bit modified CPL adder.
v(1724): The output of the 16-bit modified CPL adder.
v(110): The input.

Figure 4.15: The Delay of 32-bit modified CPL Adder.

multiplier can be divided into two pipeline stages, each of the stages contains the critical path of 16 modified CPL adders. Then the multiplication time should be less than 20 ns, which is the

specification of the multiplication.

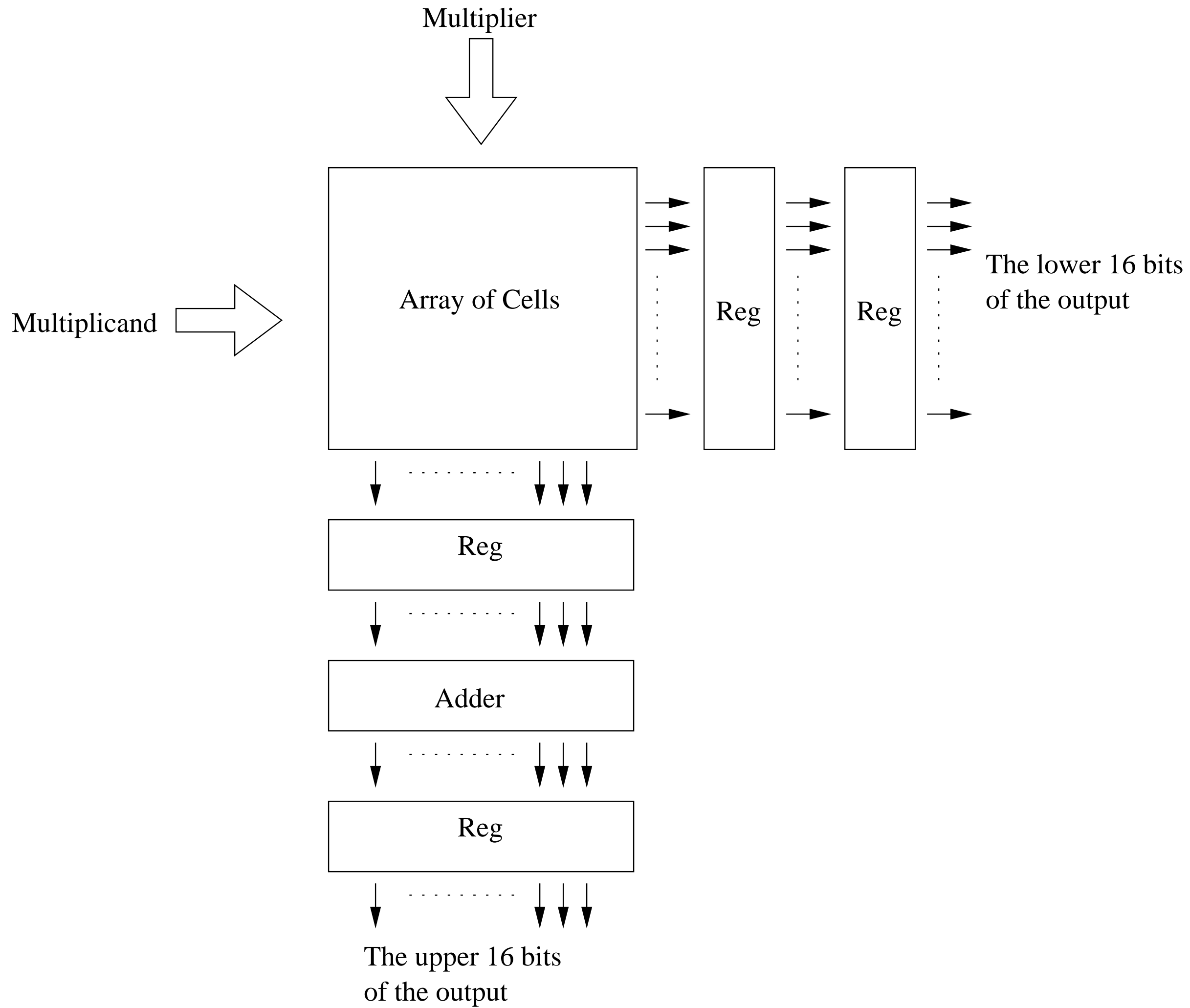


Figure 4.16: The Block Diagram of the Multiplier.

The block diagram of the multiplier is shown on Figure 4.16. The two operands of the multiplier enter the array of the multiplier cells. The array can produce the lower 16 bits of the output. The rest of the output bits of the array enter the final adder, which is the second pipeline stage. The output is stored in a register to synchronize with the other parts of Colt.

One technique is applied to the array of the multiplier cells to optimize the area and the speed. The first row of the array only passes the outputs of the AND gates to the next row, so the first row only need AND gates. The second row of the array has to add two inputs, so it only needs half adders to implement. The area and speed of an AND gate or a half adder are both less than those of a full adder.

Chapter 5

Floor Plan and Layout

The purpose of this chapter is to demonstrate the procedures and methods that were used to complete the layout of the multiplier. This includes the strategy and methods implemented on the VLSI design.

5.1 Strategy

Because both speed and area are critical in a multiplier, full-custom design is used to optimize the layout of the masks. The process of converting the schematic to the layout is complex. The role of a good design strategy is to reduce the complexity and effort of this process. This layout of the multiplier is implemented by the methods described in [Wes93].

5.1.1 Regularity

Regularity has been considered on the design of the architecture. The use of iteration to form arrays of identical cells is an illustration of the use of regularity in IC design [Wes93]. Thus it is easy to find the regularity of the layout of the multiplier.

5.1.2 Modularity

The strategy of creating well-formed modules has been applied to the Colt project. Where the notion of a well-formed module is that the module's function and interface is defined well enough to avoid any incorrect interpretation [Mus96].

5.1.3 Locality

Locality means that the internals of the modules are unimportant to the exterior interface of the modules. Modules should be placed to minimize the wiring of the global wiring. Time locality means that modules see a common clock, and hence synchronous-timing methods apply. The critical paths, if possible, should be kept within the module boundaries [Wes93].

5.2 Floor Plan and Layout

As shown in Figure 5.1, the layout of the multiplier is divided into two parts, one part for the first pipeline stage and the other for the second one. The inputs are on the western side and the northern side. The outputs are on the eastern and the northern sides. The first pipeline stage is the array of the multiplier cells, which can be further divided into 16 rows. Each row accepts one bit of one operand and all bits of the other. The last row is optimized to minimize the area as described in Section 4.6. The critical paths of the multiplier are within each row.

Registers are placed on the interface between the first and second pipeline stage and the interface between the second pipeline stage and the outputs of the multiplier. A set of extra registers are placed on the lower 16 bits of the outputs to synchronize the latency of the outputs.

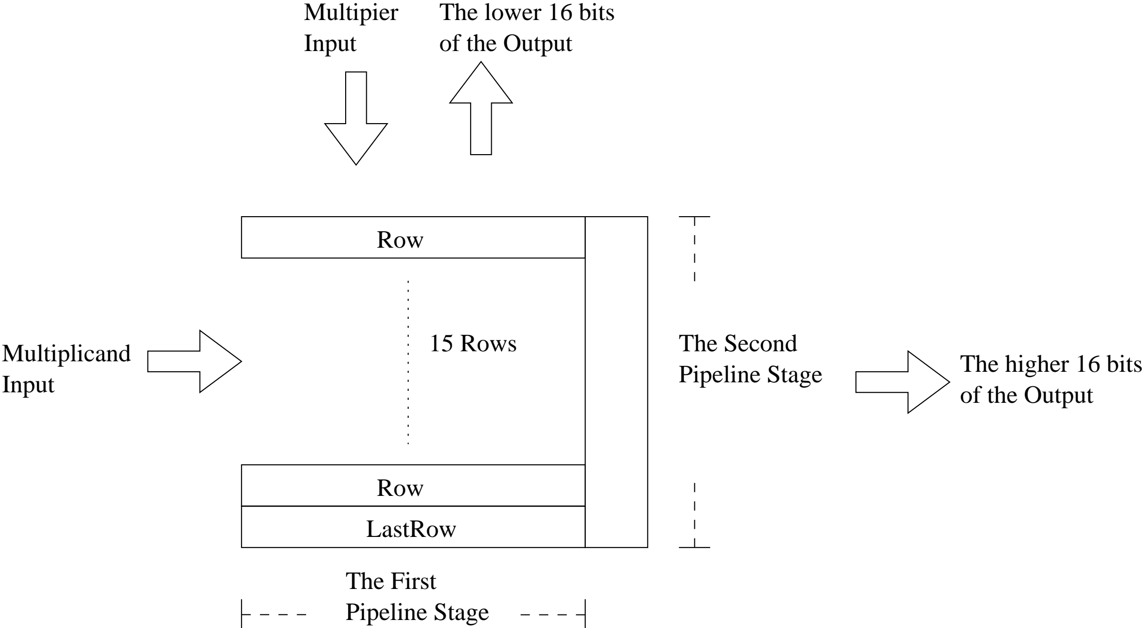


Figure 5.1: Floor Plan of Multiplier.

FirstLevel	MultCell	MultCell	MultCell	MultCell	MultCell	MultCell	MultCell

Figure 5.2: Floor Plan of a Row.

Figure 5.2 is the floor plan of a row, which consists of 7 MultCells and a FirstLevel. FirstLevel consists of two AND gates and a half adder as described in Section 4.6. Each MultCell contains two AND gates and a two-bit adder (Section 4.4).

Table 5.1 is the sheet resistances of Hp $0.5\mu m$ process. The resistance of the Poly is much greater than Metals, so the Poly used as the conductor is restricted to the cell. Metal-3 is reserved for clock and power distribution. Metal-1 and Metal-2 are used to route the signals. Figure 5.3 is the layout of the multiplier.

Table 5.1: The Sheet Resistance of HP 0.5μ Process.

Conductor	Sheet Resistance ($\Omega/\mu m^2$)
n+ diffusion	2.3
p+ diffusion	2.0
Poly	2.1
Metal 1	0.07
Metal 2	0.07
Metal 3	0.05

5.3 Verification

Two basic checks are implemented constantly on the layout to verify its correctness. First, the specified geometries in the mask layout must satisfy the design rules set by MOSIS, which is called a Design Rule Check(DRC). A summary of MOSIS's design rules are in Appendix A.2. Second, the layout of the circuit must be consistent with the schematic of the circuit. Two steps are involved to accomplish this, a circuit extraction and a layout versus schematic (LVS) check.

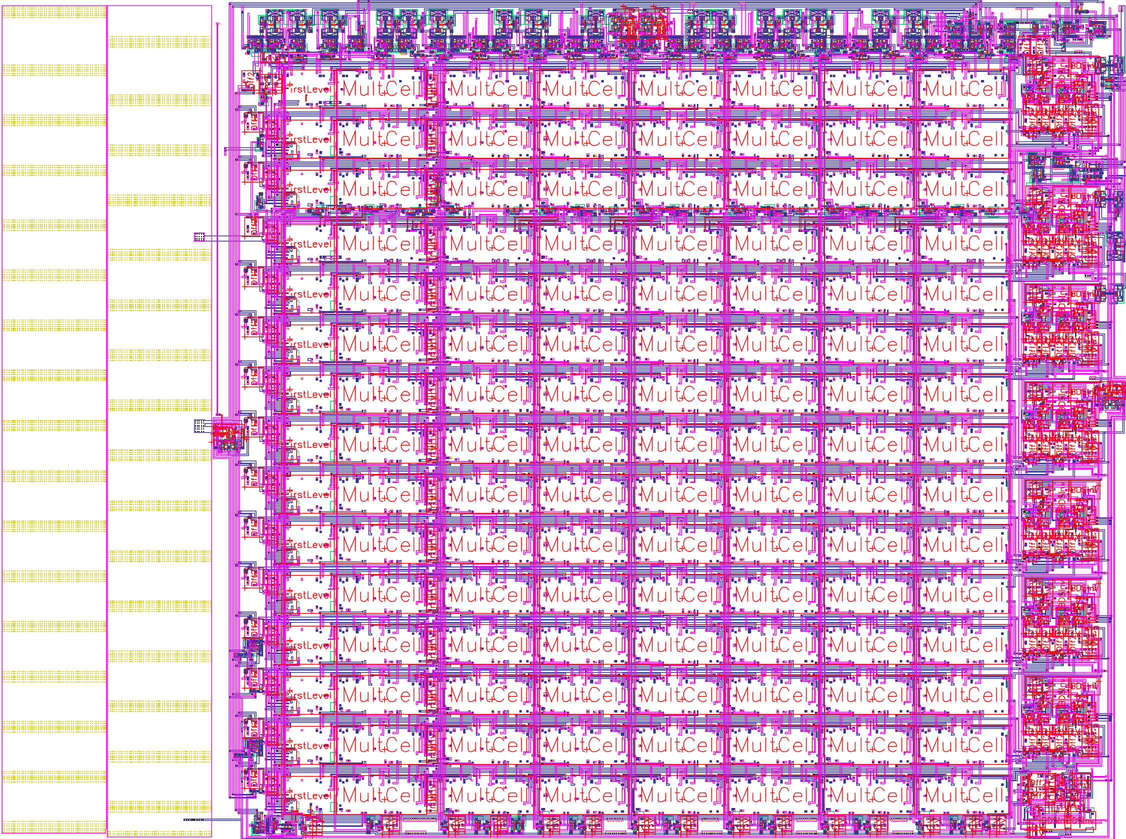


Figure 5.3: The Layout of the Multiplier.

Chapter 6

Testing and Application

It is desirable to explore the applications of the multiplier within the Colt architecture. These applications are used to verify the function of the Colt and the multiplier. As various algorithms are mapped to Colt, the next generation of Colt, Stallion, can be improved by analyzing the implementations of different applications. The high level language compiler on Colt also benefits from the created components, which can be used as the basic operators in high level language.

6.1 Testing Environment

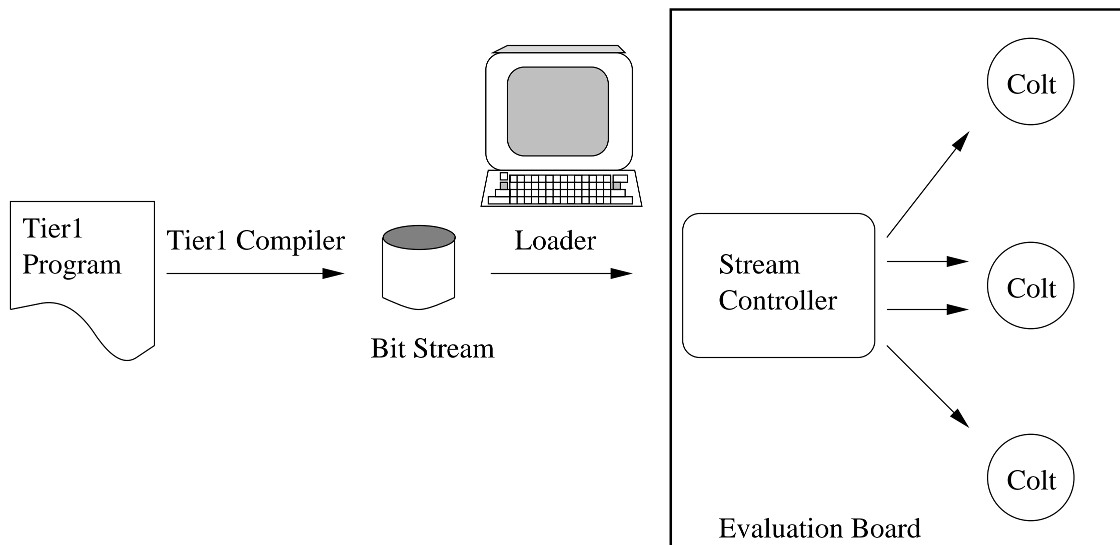


Figure 6.1: Testing Environment

The testing environment used in this project contains both hardware and software components as shown on Figure 6.1. The applications are written in a high level language, Tier1. After compiling the Tier1 program, the generated streams are loaded to the evaluation board. The stream controller on the board dispatches the streams to the specific I/O ports of the Colts.

6.1.1 Language

A high level language compiler, Tier1 [Kah97], had been developed at the time of developing the applications of Colt. The applications described in this thesis were all written in the Tier1 language. Tier1 acts as an assembler, allowing the user to have explicit control over all aspects of Colt. Tier1 also covers the detail of the configuration, which user does not specify in the program without effecting the operation of the chip. A reference on the Tier1 language can be found in [Kah97]. A Tier1 specification program can be created after the graph of the data flow is finalized. Certain features of the Colt are automated by the Tier1 complier in order to make the overall design process easier.

6.1.2 The Loader Program

The purposes of the loader are to transfer the data to the board via the PC parallel port and to trigger the board by providing clock pulses. Three arguments have to be given on the command line. They are the configuration file of Xilinx, the data stream file of the Colts and the number of the clocks to be sent to Xilinx. The loader transfers the configuration of Xilinx first. To avoid the clock of the board being interfered by the file operations, the whole data stream file of Colts is read

into a buffer before it is sent. After the reading, the program transfer the data from the buffer to the board. A clock pulse has to be provided as each byte of the data is being transfered.

6.1.3 Evaluation Board

Figure 6.2: The Block Diagram of Colt Evaluation Board

The verification is performed on an evaluation board. Figure 6.2 shows the block diagram of Colt Evaluation Board, which consists of one Xilinx 4010 and four Colts. The Xilinx 4010 serves as the stream controller. The network of Colts is completely connected, allowing either Colt to interact with any other Colt on the board. Although 4 Colts are on the board, the applications on

this research only need one Colt. The rest of the chips are reserved for the future evaluation.

6.1.4 Xilinx Configuration

Because various applications are being developed on the Colt, the configurations of the Colt are versatile. For example, one application may need only one stream tunneling through a Colt, but another application may need two streams entering a Colt. Also, a port may be used as an input port, an output port or even a bi-directional port. The stream controller must be able to be configured for different situations. An FPGA can easily satisfy the reconfigurability of the stream controller.

```
FF FF FF FF
FF FF FF FF
FF FF FF FF
FF
39 CD 81 FF
39 CD 91 FF
38 00 91 FF
38 00 A1 FF
38 00 F1 FF
38 01 01 FF
3D CD C1 FF
3D CD D1 FF
B0 00 01 FF
B0 00 01 FF
B0 00 11 FF
B0 00 11 FF
```

Figure 6.3: A Data Stream File

Figure 6.3 shows the content of a data stream file. This configuration accepts two operands and sends the multiplications of these two operands to two output ports. The first four lines are used

to reset the Colts and to synchronize the data. Only the first five nibbles of each line are sent to the ports. The rest of the line will be disregarded. The first nibble specify the four control bits of the ports. They are program bit, writing bit, transmission bit and ready bit from the highest bit to the lowest bit respectively. The other four nibbles are the data bits. The lines of even number are the stream into port 0. The lines of odd number are the stream into port 1.

Before the configuration of Colt, a reset must be low for at least two clock cycles. It is necessary to inject 20-bit wide programming word into Colt, but the parallel port can transfer a byte only per clock cycle. Three cycles are needed to transfer 20 bits by the parallel port, and one extra cycle is used as the clock to broadcast the data into Colts. Total eight clock cycles are needed to inject the data into two ports simultaneously.

6.2 Application: Matrix Multiplication

The first application of the Colt and the multiplier, of course, is the multiplier alone. This application also is the basic configuration to verify the multiplier before any further exploration of applications. Four ports are needed for the verification of the multiplier. Two ports are the input ports and the other two are the output ports. Each programming stream enters one input data port, through the crossbar, the multiplier, and goes to one output port via the crossbar again. The data stream is catenated at the end of the programming stream and processed by the multiplier. Figure 6.4 shows the Tier1 source code of the multiplication.

Several other applications about Colt and its multiplier such as dot multiplication, floating-point multiplication and factorial function can be found in [Bit97a]. Matrix multiplication will be

```
//
// Simple multiplier example.  Routes two streams through the multiplier
// and out to the two output ports
//

#define HIPORT 1
#define LOPORT 2

ports
inhi = 1; // low byte input
inlo = 2; // high byte input
outhi = 5; // output high byte
outlo = 6; // output low byte
end ports;

// Input stream for high byte
stream multhi (in inhi, out outhi)

port inhi => mult at HIPORT;
mult at HIPORT => port outhi;

end stream;

// Input stream for low byte
stream multlo (in inlo, out outlo)

port inlo => mult at LOPORT;
mult at LOPORT => port outlo;

end stream;
```

Figure 6.4: Tier1 Source of the Multiplication

discussed in this chapter.

6.2.1 Overview

Matrix multiplication is ubiquitous in many computer applications such as circuit analysis, image processing and mathematical applications. The multiplication of two matrixes $R = AB$ can be evaluated by the following equation.

$$\begin{bmatrix} r_{11} & \cdots & r_{1l} \\ \vdots & \ddots & \vdots \\ r_{m1} & \cdots & r_{ml} \end{bmatrix} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} b_{11} & \cdots & b_{1l} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nl} \end{bmatrix} \tag{6.1}$$

where r_{ij} can be expressed by

$$r_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{6.2}$$

The number of columns of the A matrix must be equal to the number of rows of the B matrix. Each item of the R matrix requires n multiplications. For an $m \times l$ R matrix, total lmn multiplications are required. Since there is only one multiplier in Colt, the bottleneck would be the multiplier. For the Equation 6.1, at least lmn cycles are necessary if the multiplier is always available.

There are many ways to implement the matrix multiplication. Which method is optimal depends on the available resources and the size of the matrix. The method with minimum resources will be assumed. Figure 6.5 shows a possible implementation of matrix multiplication. The outputs of the multiplier is accumulated by an IFU. The purpose of the control block is to reset the accumulator.

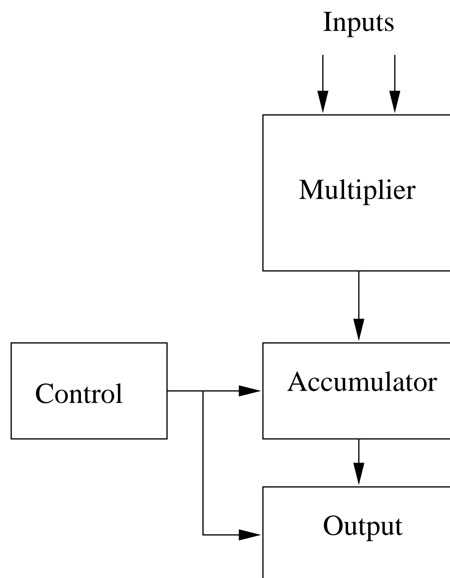


Figure 6.5: Matrix Multiplication

6.2.2 Data Flow Graph Mapping

Table 6.1: The ALU Term Configuration

Function	P	G	R	Comment
Decrement Left	3	0	9	Carry = 1; Out=Left-1
Addition	6	8	6	Out=Left+Right
Passing Left	C	0	A	Out=Right

Figure 6.6 shows the actual implementation. It is the programmer's responsibility to arrange the data stream of the inputs and to receive the data stream of the output in the correct order under this implementation. This implementation only needs three IFUs. The first IFU only passes the data stream to the other IFUs on the same column.

The second IFU is the counter to reset the accumulator to zero. The left register is initialized

to zero and latches the first data word of the stream, which is the width of A matrix or the length of B matrix. The output is looped back to its own left register. ALU is configured as the function of decreasing left as shown on Table 6.1. The inverse of the carry out is used as the conditional flag, which is changed from zero to one as the output of ALU is decreased to zero. If the conditional flag is true (1), the output is the output of ALU. If not, the output is the value of the right register. The register should contain the value to set the left register at this time. The stream must contain the width of A matrix then. If A matrix is $m \times n$ and B matrix is $n \times l$, then total $(n+1)lm$ data words are needed to complete the matrix multiplication. For example, if A and B matrixes are both 3×3, then any item in R matrix requires 3 multiplications and 3 additions. The total number of the data words is $(3+1) \times 3 \times 3 = 36$. Figure 6.7 shows the data stream entering Colt.

The third IFU is the accumulator. ALU is configured to the function of addition as shown on Table 6.1. The output is looped back to the left register, too. It receives the conditional output flag, which is the carry out flag, from the second IFU via a skip bus. If the conditional flag is true, the left register is reset to zero.

The last IFU is the output control. The valid bit originates from the carry out of the second IFU via the skip bus. If the valid bit is true, the output of this IFU is the needed data. The ALU is configured to the function of passing the left input as shown on Table 6.1. The output of this IFU is connected to the output data port.

This configuration gives much flexibility. Because the number of the required accumulations is specified in the data stream, this configuration can do any matrix multiplication if the width of A matrix and the length B matrix do not exceed $2^{15} = 32768$ and an overflow dose not happen. Also,

the dimensions of the two matrixes can be changed dynamically. This configuration only need 3 IFUs which is 19% of reconfigurable resources of the Colt.

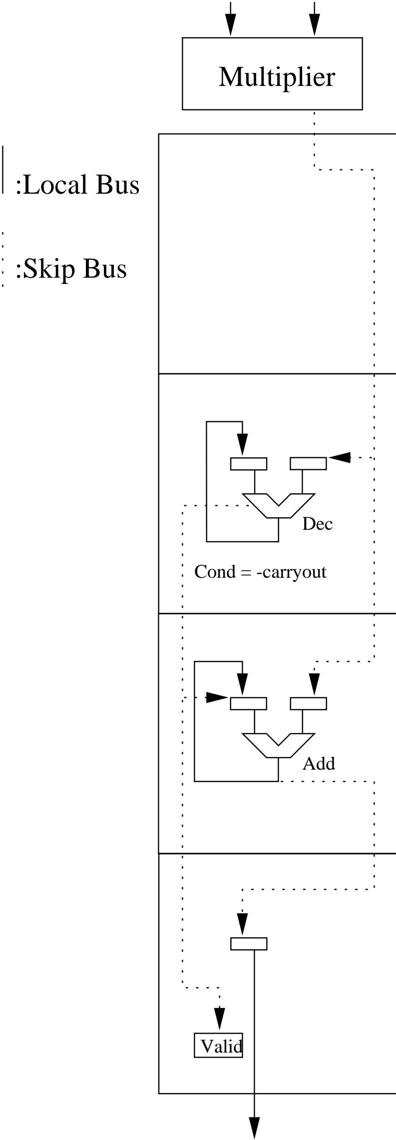


Figure 6.6: Configuration for Matrix Multiplication

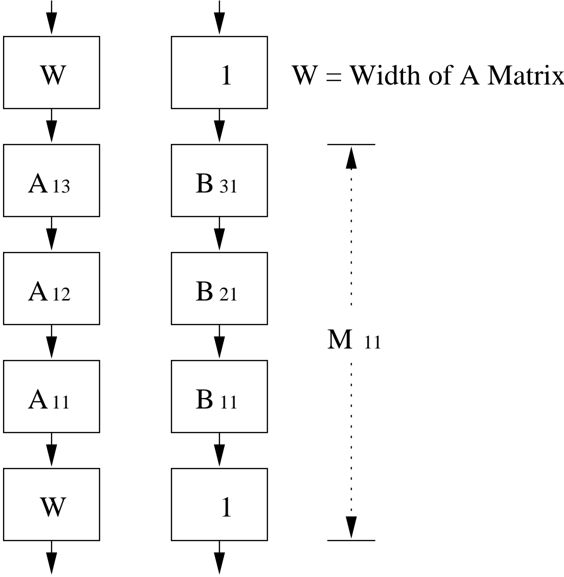


Figure 6.7: Data Stream Example

Chapter 7

Conclusion

7.1 Result

Table 7.1: Dimension and Current of the Multiplier.

Height(μm)	603.6
Width (μm)	811.05
Area (mm^2)	0.5
Average Current (mA)	15
Peak Current (mA)	90
Average Power (mW)	49.5

Table 7.1 shows the square dimensions, the average current and power drain of the multiplier. The speed of the multiplier should be within 50 MHz according to the simulation.

The function of the multiplier has been verified as discussed on Chapter 6. Due to a bug in IFUs, the latch function of IFUs is not working, so the matrix multiplication is not working. However, the concept has been verified on simulation.

7.2 Future Enhancement

Since the Colt platform is relatively new, there is much room for additional work and improvement. It would be useful to design the next generation multiplier with the ability to performed

signed multiplication. The speed of the Stallion is targeted at 100 MHz, so the speed of the multiplier has to be doubled then.

7.2.1 Signed Multiplication

It is recommended that signed multipliers embedded in the next generation of Colt [Bit97a]. A powerful algorithm for signed-number multiplication is Booth's algorithm as discussed in Subsection 3.2.3.

The physical implementation of Booth's algorithm needs a Booth encoder for the operands, an array of multiplier cells and a final adder as shown in Figure 7.1.

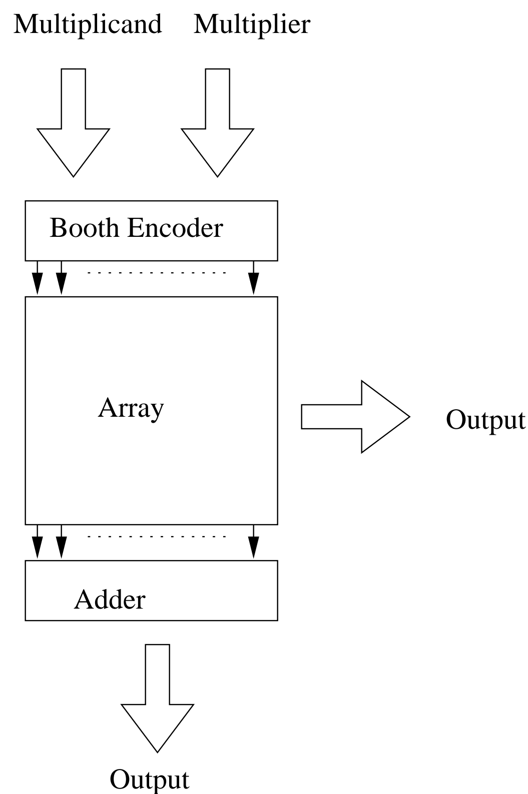


Figure 7.1: Implementation of Booth Multiplier

7.2.2 Speed

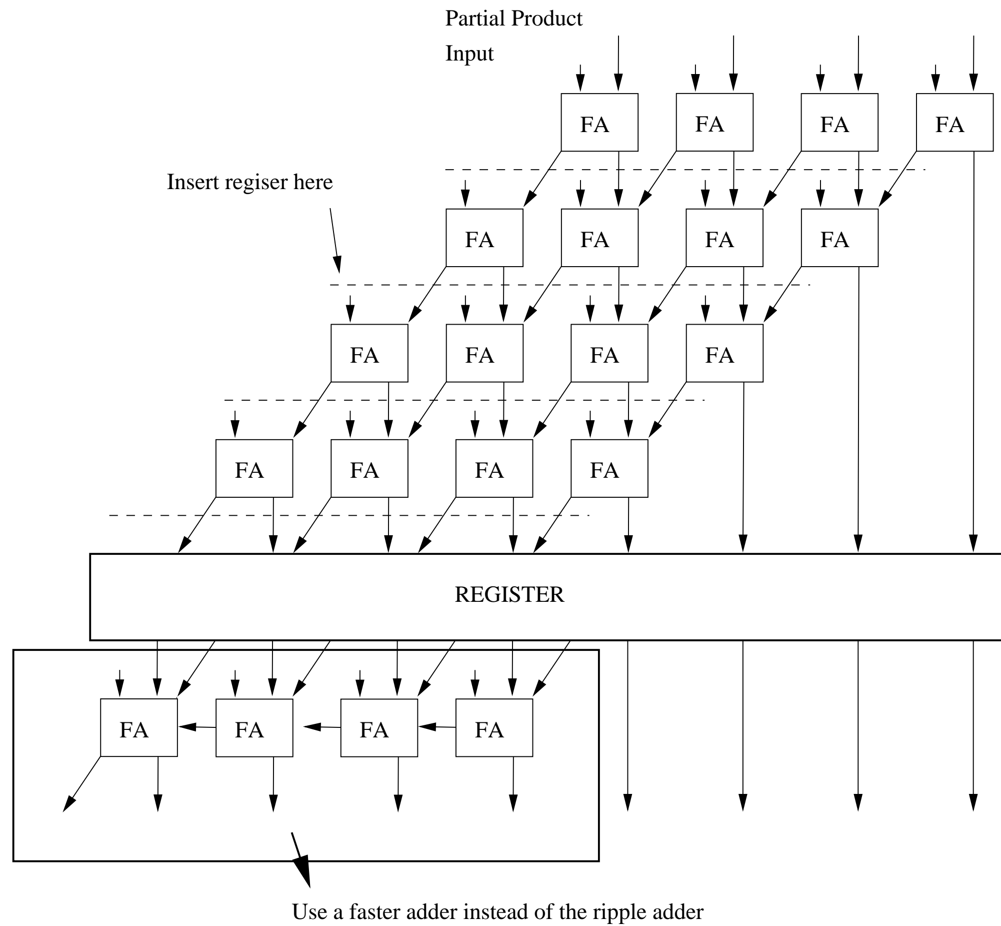
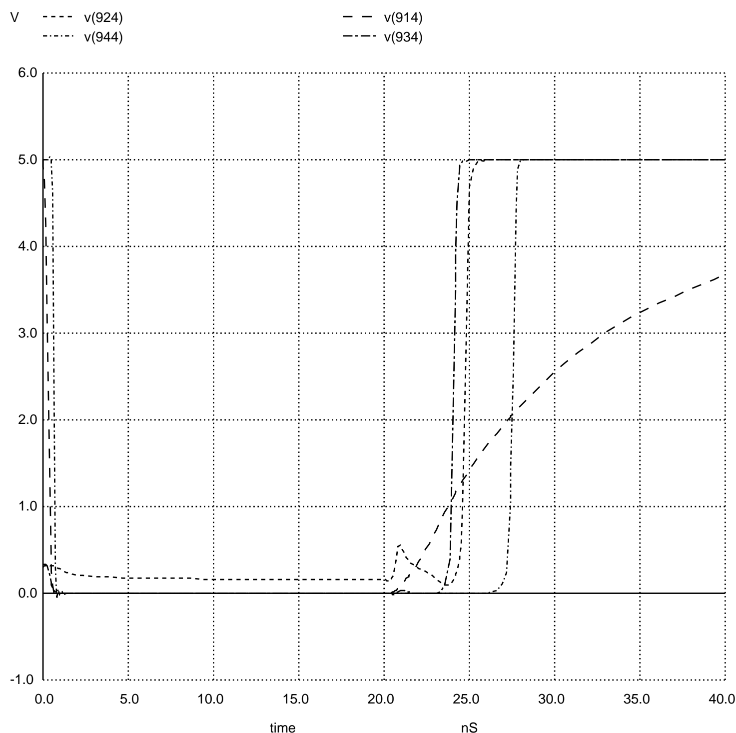


Figure 7.2: Speed Improvement of Multiplier Architecture

It is necessary to increase the speed of the multiplier for the next generation of Colt. Because the multiplier is pipelined, it is easy to make a faster multiplier by inserting more pipeline stages. Figure 7.2 shows a possibility to insert one more pipeline stage into the multiplier. Many fast adders such as conditional-sum adders [Sla60] and carry-select adders [Bed62] are suitable for the last adder.

It is still possible to enhance the speed of the multiplier without extra pipeline stages. By applying Booth bit-pair recoded method as discussed in Subsection 3.2.3, an n-bit multiplier will generate at most $n/2$ summands and that uniformly handles the signed-operand case. For a non-pipelined multiplier, the speed of this implementation should be at least 50 MHz. For a two-stage multiplier, it should be over 100 MHz.

7.2.3 Adder



v(914): Transmission-Gate Adder
v(924): Modified CPL Adder
v(934): CMOS Pass-Transistor Adder
v(944): Transmission-Gate Adder with Output Buffer

Figure 7.3: Comparison of Four Adders

It is important to take into account the problems of noise margins and speed degradation. As the process of fabrication continues shrinking, these factors become important to ensure the circuit work. These are caused by mismatches between the input signal levels and the logic threshold voltage of the NMOS gates. The adder can be improved by replacing the NMOS pass-transistor logic with complementary pass-transistor logic. Figure 7.3 shows the outputs of 4 8-bit adders.

REFERENCES

[Asa90] C. Asato, C. Ditzen and S. Dholakia, "A Data-Path Multiplier with Automatic Insertion of Pipeline Stages," IEEE Journal of Solid-State Circuits, Vol. 25, April 1990.

[Bed62] O. J. Bedrij, "Carry-select adder,"IRE Transactions on Electronic Computers, Vol. EC-11, June 1960.

[Bit97a] R. A. Bittner, Jr., "Design and VLSI Implementation of a High Speed Data Flow DSP Computing System," Ph.D. Dissertation, Virginia Polytechnic Institute and State University, 1997.

[Bit97b] R. Bittner, P. Athanas, "Wormhole Run-time Reconfiguration," ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 79-85, February, 1997.

[Boo51] A. D. Booth, "A Signed Binary Multiplications Technique," Quarterly Journal of Mechanical and Applied Math, vol. 4, 1951.

[Kah97] B. C. Kahne, "A Genetic Algorithm-Based Place-and-Route Compiler for a Run-time Reconfigurable Computing System," Master's thesis, Virginia Polytechnic Institute and State University, 1997.

[Che96] M. Cherbaka, "Verification and Configuration of a Run-time Reconfigurable Custom Computing Integrated Circuit for DSP Applications," Master's thesis, Virginia Polytechnic Institute and State University, 1996.

[Chu87] K. M. Chu and D. L. Pulfrey, " A comparison of CMOS circuit techniques: Differential Cascade Voltage Switch Logic versus Conventional Logic," IEEE Journal of Solid-State Circuits, Vol. SC-22, 1987

[Gei90] R. L. Geiger, P. E. Allen, N. R. Strader, *VLSI Design Techniques for Analog and Digital Circuits,* McGraw-Hill, 1990.

[Ham90] V. H. Hamacher, Z. G. Vranesic and S. G. Zaky, *Computer Organization,* McGraw-Hill, 1990.

[Hat86] M. Hatamian and G. L. Cash, "A 70-MHz 8-bit X 8-bit Parallel Pipelined Multiplier in 2.5-μm CMOS," IEEE Journal of Solid-State Circuits, Vol. SC-21, No. 4, August 1986.

[Hel84] L. G. Heller, W. R. Griffin, J. W. Davis, and N. G. Thoma, "Cascade Voltage Switch Logic: a Differential CMOS Logic Family," Proceeding of the IEEE International Solid State Circuits Conference, February 1984.

[Hwa93] Kai Hwang, *Advanced Computer Architecture,* McGraw-Hill, 1993.

[Got92] G. Goto, T. Sato, M. Nakajima, and T. Sukemura, " A 54 X 54 Regularly Structured Tree Multiplier," IEEE Journal of Solid-State Circuits, Vol. 27 No. 9, September 1992.

[Mus96] M. D. Musgrove, "VLSI Implementation of a Run-time Reconfigurable Custom Computing Integrated Circuit," Master's thesis, Virginia Polytechnic Institute and State University, 1996.

[Pet95] Russell J. Petersen and Brad L. Hutchings, "An Assessment of the Suitability of FPGA-Based Systems for use in Digital Signal Processing," 5 th International Workshop on Field Programmable Logic and Applications, Oxford, England, August 1995.

[Ohk95] N. Ohkubo, M. Suzuki, T. Shinbo, T. Yamanaka, A. Shimizu, K. Sasaki, and Y. Nakagome, "A 4.4 ns CMOS 54 x 54-b Multiplier Using Pass-Transistor Multiplexer," IEEE Journal of Solid-State Circuits, Vol. 30 No. 3, March 1995.

[San89] M. R. Satoro and M. A. Horowitz, "SPIM: A Pipelined 64x64-bit Iterative Multiplier," IEEE Journal of Solid-State Circuits, Vol. 24 No. 2, April 1989.

[Sla60] J. Slansky, "Conditional-Sum Addition Logic," IRE Transactions on Electronic Computers, Vol. EC-9, June 1960.

[Ste90] C. C. Stearns and P. H. Ang, "Yet Another Multiplier Architecture," IEEE Custom Integrated Circuits Conference, 1990.

[Tan95] W. Tanner, *The MOSIS User Manual 4.0,* University of Southern California, 1995.

[Vui82] J. Vuillemin, "A Very Fast Multiplication Algorithm for VLSI Implementation," INTEGRATION, the VLSI Journal, 1983.

[Wes93] N. H. E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design: a Systems Perspective,* 2nd ed., Addison-Wesley, 1993.

[Wal64] C. S. Wallace, "A Suggestion for a Fast Multiplier," IEEE Transactions on Electronic Computers, February 1964.

[Xil94] Xilinx, Inc., *The Programming Logic Data Book,* 1994.

[Yan90] K. Yano, T. Yamanaka, T. Nishida, M. Saito, K. Shimohigashi, and A. Shimizu, "A 3.8-ns CMOS 16x16-b Multiplier Using Complementary Pass-Transistor Logic," IEEE Journal of Solid-State Circuits, Vol. 25 No. 2, April 1990.

Appendix A

HP05 Design Process

A.1 MOSIS Parametric Measurements

```
                     MOSIS PARAMETRIC TEST RESULTS

        RUN: N65A                                          VENDOR: HP-NID
  TECHNOLOGY: SCN05H                                 FEATURE SIZE: 0.5 microns


INTRODUCTION: This report contains the lot average results obtained by MOSIS
              from measurements of MOSIS test structures on each wafer of
              this fabrication lot. SPICE parameters obtained from similar
              measurements on a selected wafer are also attached.

COMMENTS: Hewlett Packard CMOS14TB.

TRANSISTOR PARAMETERS   W/L      N-CHANNEL P-CHANNEL  UNITS

 MINIMUM                0.9/0.60
  Vth                                0.71     -0.90  Volts

 SHORT                  15/0.60
  Vth                                0.64     -0.90  Volts
  Vpt                               10.0     -10.0   Volts
  Vbkd                              11.3      -9.4   Volts
  Idss                             379      -191     uA/um

 WIDE                   15/0.60
  Ids0                              13.7       3.5   pA

 LARGE                  5.4/5.4
```

Vth	0.72	-0.96	Volts
Vjbkd	11.5	-10.0	Volts
Ijlk	-17.1	-2.4	pA
Gamma	0.62	0.50	V^0.5
Delta length (L_eff = L_drawn-DL)	0.06	0.09	microns
Delta width (W_eff = W_drawn-DW)	0.36	0.35	microns
K' (Uo*Cox/2)	69.3	-21.8	uA/V^2

COMMENTS: Delta L varies with design technology as a result of the different mask biases applied for each technology. Please adjust the delta L in this report to reflect the actual design technology of your submission.

Design Technology	Delta L
SCN_SUBM (lambda=0.3), CMOSH, HP_EMOS14TB	no adjustment
SCN (lambda=0.35)	add 0.1 um

FOX TRANSISTORS	GATE	N+ACTIVE	P+ACTIVE	UNITS
Vth	Poly	>15.0	<-15.0	Volts

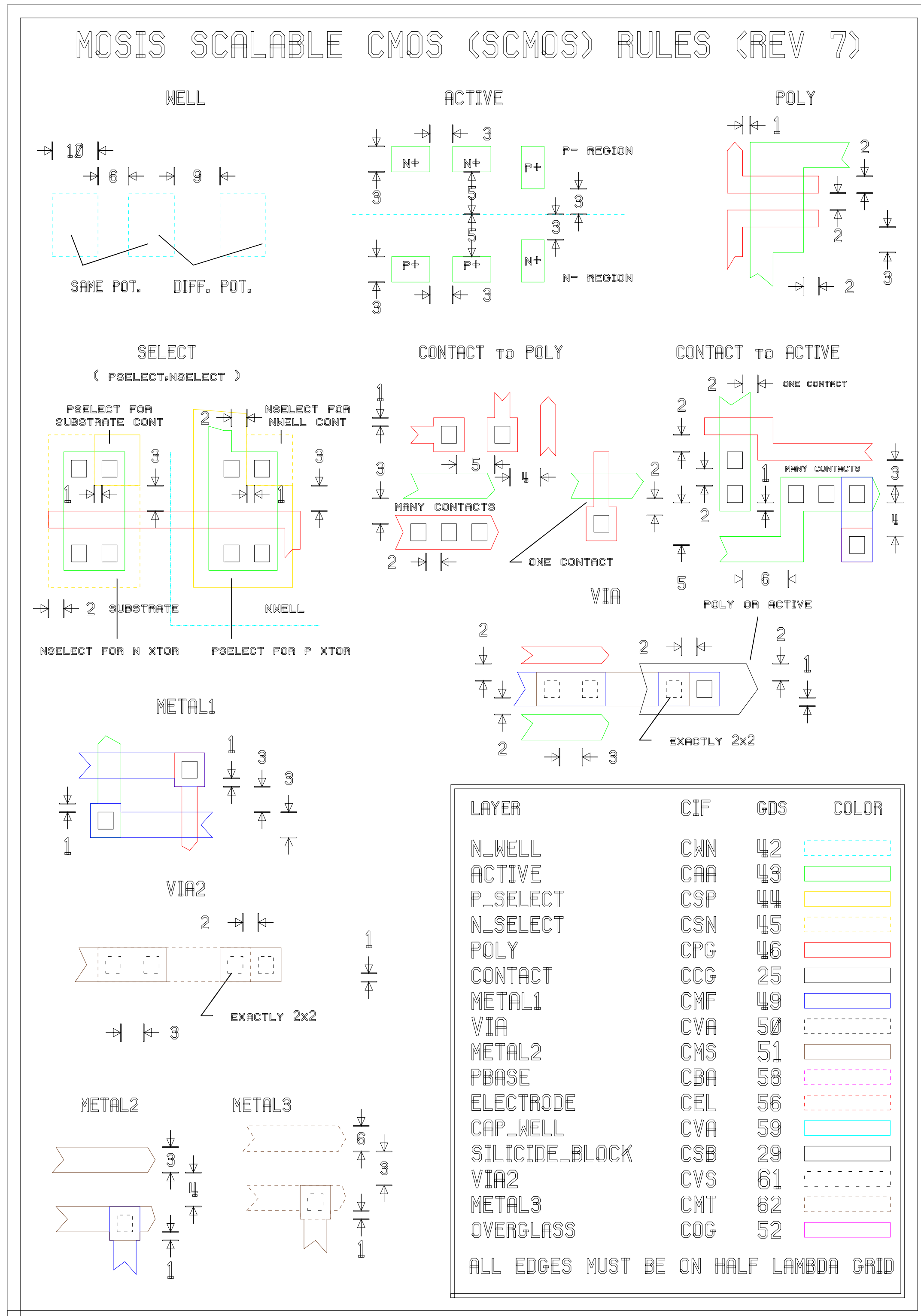
PROCESS PARAMETERS	N+DIFF	P+DIFF	POLY	METAL1	METAL2	METAL3	UNITS
Sheet Resistance	2.0	2.1	1.9	0.07	0.07	0.05	ohms/sq
Width Variation (measured - drawn)			-0.06	0.14	0.04	-0.35	microns
Contact Resistance	2.8	2.8	3.0		0.77	1.27	ohms
Gate Oxide Thickness	96						angstroms

CAPACITANCE PARAMETERS	N+DIFF	P+DIFF	POLY	METAL1	METAL2	METAL3	UNITS
Area (substrate)	538	937	92	46	24	25	aF/um^2
Area (poly)				55	17	10	aF/um^2
Area (metal1)					39	14	aF/um^2

```
 Area (metal2)                                                       32      aF/um^2
 Area (N+active)                   3595                                      aF/um^2
 Area (P+active)                   3390                                      aF/um^2
 Fringe (substrate)     205     257                                          aF/um
 Fringe (N+active)                  100                                      aF/um


CIRCUIT PARAMETERS                            UNITS
 Inverters                    K
  Vinv                       1.0        1.30  Volts
  Vinv                       1.5        1.44  Volts
  Vol (100 uA)               2.0        0.20  Volts
  Voh (100 uA)               2.0        3.04  Volts
  Vinv                       2.0        1.54  Volts
  Gain                       2.0      -17.63
 Ring Oscillator
  DIV4 (31-stage,3.3V)               139.23  MHz

COMMENTS:
=================================================================

SPICE Model Parameters for Submicrometer Technologies

Completely new data gathering and parameter optimization strategies
for submicrometer technologies are under development at MOSIS. As
these strategies are tested and proven to be accurate, new sets of
submicrometer SPICE parameters will be made available. During the
transition, and for some period thereafter, Level=3 parameters will
continue to be provided.  Unfortunately, for the reasons stated in the
file submicron-spice-parameters.inf (located in the MOSIS FTP access:
ftp/pub/mosis/info), we cannot guarantee the accuracy of these Level=3
parameters in submicrometer technologies. Designers should begin
thinking about making a transition away from Level=3 model parameters.

MOSIS will continue offering BSIM model parameters, Level=4, along
with the Level=3 parameters. We will be supplying BSIM3v3 in the near
future to improve simulation performance in submicrometer
technologies.

=================================================================
```

```
 N65A SPICE LEVEL3 PARAMETERS

.MODEL CMOSN NMOS LEVEL=3 PHI=0.700000 TOX=9.6000E-09 XJ=0.200000U TPG=1
+ VTO=0.7118 DELTA=2.3060E-01 LD=2.9830E-08 KP=1.8201E-04
+ UO=506.0 THETA=1.9090E-01 RSH=1.8940E+01 GAMMA=0.6051
+ NSUB=1.4270E+17 NFS=7.1500E+11 VMAX=2.4960E+05 ETA=2.5510E-02
+ KAPPA=1.8530E-01 CGDO=9.0000E-11 CGSO=9.0000E-11
+ CGBO=3.7295E-10 CJ=6.02E-04 MJ=0.805 CJSW=2.0E-11
+ MJSW=0.761 PB=0.99
* Weff = Wdrawn - Delta_W
* The suggested Delta_W is 3.5700E-07
.MODEL CMOSP PMOS LEVEL=3 PHI=0.700000 TOX=9.6000E-09 XJ=0.200000U TPG=-1
+ VTO=-0.9016 DELTA=4.2020E-01 LD=4.3860E-08 KP=4.1582E-05
+ UO=115.6 THETA=3.7990E-02 RSH=9.0910E-02 GAMMA=0.4496
+ NSUB=7.8780E+16 NFS=6.4990E+11 VMAX=2.3130E+05 ETA=2.8580E-02
+ KAPPA=9.9270E+00 CGDO=9.0000E-11 CGSO=9.0000E-11
+ CGBO=3.6835E-10 CJ=9.34E-04 MJ=0.491 CJSW=2.41E-10
+ MJSW=0.222 PB=0.90
* Weff = Wdrawn - Delta_W
* The suggested Delta_W is 3.4860E-07

==================================================
```

A.2 MOSIS Design Rules


MOSIS SCALABLE CMOS (SCMOS) RULES (REV 7)
WELL
10
6
9
SAME POT.
DIFF. POT.
ACTIVE
N+
P+
P- REGION
N- REGION
POLY
SELECT
(PSELECT,NSELECT)
PSELECT FOR SUBSTRATE CONT
NSELECT FOR NWELL CONT
SUBSTRATE
NWELL
NSELECT FOR N XTOR
PSELECT FOR P XTOR
CONTACT TO POLY
MANY CONTACTS
ONE CONTACT
CONTACT TO ACTIVE
ONE CONTACT
MANY CONTACTS
POLY OR ACTIVE
VIA
EXACTLY 2x2
METAL1
VIA2
EXACTLY 2x2
METAL2
METAL3
LAYER CIF GDS COLOR
N_WELL CWN 42
ACTIVE CAA 43
P_SELECT CSP 44
N_SELECT CSN 45
POLY CPG 46
CONTACT CCG 25
METAL1 CMF 49
VIA CVA 50
METAL2 CMS 51
PBASE CBA 58
ELECTRODE CEL 56
CAP_WELL CVA 59
SILICIDE_BLOCK CSB 29
VIA2 CVS 61
METAL3 CMT 62
OVERGLASS COG 52
ALL EDGES MUST BE ON HALF LAMBDA GRID

Appendix B

Various Tools and Source Codes

B.1 th2spi.awk

“th2spi.awk” is a program written in AWK to convert the test patterns to the format of spice by “pwl” function. To use this program, edit a input file “name.2spi” as described below. Then enter the command:

```
awk -f th2spi.awk < name.2spi >> name.cir
```

where name.cir is the file containing the description of the circuit. The following is the script file “th2spi.awk”.

```
# This program is to generate the pattern of spice pwl function.
# The following is a example for the input file of this program.
#
# Delay Time = 0.5
# Falling Time = 0.5
# Half Cycle = 20
# Voltage = 5
# State Trans a 2 0 = 0 0 1 1 0 0 1 1 1
#
# The following is the result of the above input file.
#
#Va 2 0 PWL( 0 5  0.5ns 0 20ns 0  20.5ns 0 40ns 0
#+ 40.5ns 5 60ns 5  60.5ns 5 80ns 5  80.5ns 0 100ns 0  100.5ns 0 120ns 0
#+ 120.5ns 5 140ns 5  140.5ns 5 160ns 5  160.5ns 5 180ns 5 )

$1 =="Delay" && $2 == "Time" && $3=="=" { delay = $4 }
$1 =="Half" && $2 == "Cycle" && $3=="=" { cycle = $4 }
$1 =="Voltage" && $2=="=" { volt = $3 }
```

```
$1 =="State" && $2 == "Trans" {
  time = 0 ;
  gap = time + delay ;
  printf "V%s %s %s PWL( 0 ",$3, $4, $5 ;
  if( S7 ) printf "0" ; else
    printf "%s ",volt ;
  for(  i=7; i<=NF; i++)
    {
      printf " %sns ", gap;
      if( $i ) printf "%s ", volt; else
        printf "0 ";
      time = time + cycle ;
      gap = time + delay ;
      printf "%sns ",time ;
      if( $i ) printf "%s ", volt; else
        printf "0 ";
      if(i%4 ==0) printf "\n+"
    }
  printf ")\n"
    }
$1 == "End" { printf ".END" }
```

B.2 mx

mx is a program to convert the output of TIER1 to the data streams of Colts.

```
#!/usr3/local/bin/perl
# Program for generating the stream of matrix multiplication
# on Colt.
# Three files are needed. They are port0.pwl, port1.pwl and matrix.
# port0.pwl and port1.pwl are the files generated by tier1.
# matrix is the file to specify the input pattern. Six files are
# getnerated: mx.dat, mx.hex, ports, result, port0.dat and port1.dat.
# mx.hex is the file to load.
# result is the correct output of Colt.
# port0.dat and port1.dat are the files for simulation

$p0 = "port1.pwl";
$p1 = "port2.pwl";
$data0 = 'port0.dat';
$data1 = 'port1.dat';
```

```
$pattern = "mx.dat";
$matrix = "matrix";
$result = "result";
$ports = "ports";
$hex = "mx.hex";
$prog_bits = "0011";
$dummy = "00110000000000000000";
$dummydata = '0' . "0" x 16;
open(PORT0, $p0) || die "$p0 doesn't exist";
open(PORT1, $p1) || die "$p1 doesn't exist";
open(D0,"> $data0") || die "Can't create $data0";
open(D1,"> $data1") || die "Can't create $data1";
open(PATTERN, "> $pattern") || die "Can't create $p";

# This part scans port0.pwl and port1.pwl.
# Put the pattern to the file alternatively.
while(<PORT0>){
    if(/[10]{16}/){
$s = $prog_bits . $&;
$d = '0' . $&;
# put data for port0
print PATTERN "$s \n";
print D0 "$d \n";
$found = 1;
    }
    $_ = <PORT1>;
    if(/[10]{16}/){
$s = $prog_bits . $&;
$d = '0' . $&;
# put dat for port1
print PATTERN "$s \n";
print D1 "$d \n";
    }elsif($found == 1){
# put dat for port1
print PATTERN "$dummy \n";
print D1 "$dummydata \n";
    }
    $found = 0;
}
close PORT0;
close PORT1;
```

```
# Deal with matrix file.
# Generating the data stream to Colt
open(MX, "$matrix");
$mb = 0;
$length_ma = 0;
$length_mb = 0;

# Generate two matrixes.
while(<MX>){
    if($_ ne "\n"){
if($mb == 0){
    @tmp = split;
    $width_ma = @tmp;
    $ma[$length_ma++] = [@tmp];
}else{
    @tmp = split;
    $width_mb = @tmp;
    $mb[$length_mb++] = [@tmp];
}
    }else{
$mb = 1;
    }
}
if($width_ma != $length_mb){
    die "The width of MA is not equal to the length of MB.";
}

# Generate the result file
open(RESULT, "> $result") || die "Can't create the file result";
for($r = 0; $r < $length_mb; $r++){
    for($c = 0; $c < $width_ma; $c++){
$result = 0;
for($i = 0; $i < $length_mb; $i++){
    $result = $result + hex($ma[$r][$i]) * hex($mb[$i][$c]);
}
$result = sprintf "%4.0x ", $result;
print RESULT "$result ";
    }
    print RESULT "\n";
}
```

```
close RESULT;

# Convert two matrixes to binary format.
for($r = 0; $r < $length_ma; $r++){
    for($c = 0; $c < $width_ma; $c++){
$ma[$r][$c] = &hex2bin($ma[$r][$c]);
print "$ma[$r][$c] ";
    }
    print "\n";
}
for($r = 0; $r < $length_mb; $r++){
    for($c = 0; $c < $width_mb; $c++){
$mb[$r][$c] = &hex2bin($mb[$r][$c]);
print "$mb[$r][$c] ";
    }
    print "\n";
}

# Produce the data stream.
$invalid = "1001" . "1" x 16;
$data_bits = "1011";
$l = sprintf "%x", $width_ma;
$ldata = '1' . &hex2bin($l);
$l = $data_bits . &hex2bin($l);
$one = $data_bits . "0" x 15 . "1";
$onedata = '1' . "0" x 15 . '1';
$zero = $data_bits . "0" x 16;
$prog_zero = $prog_bits . "0" x 16;
# print data for port0 and port1
print PATTERN "$prog_zero\n" x 2;
print PATTERN "$invalid\n" x6;
print PATTERN "$zero\n" x 2;
print D0 "$dummydata\n" x 4;
print D1 "$dummydata\n" x 4;
for($r = 0; $r < $length_mb; $r++){
    for($c = 0; $c < $width_ma; $c++){
# print width data for port0
print PATTERN "$l\n";
print D0 "$ldata\n";
# print a one for port1
print PATTERN "$one\n";
```

```
print D1 "$onedata\n";
for($i = 0; $i < $length_mb; $i++){
    $a = $data_bits . $ma[$r][$i];
    $adata = '1' . $ma[$r][$i];
    $b = $data_bits . $mb[$i][$c];
    $bdata = '1' . $mb[$i][$c];
    # print data for port0
    print PATTERN "$a\n";
    print D0 "$adata\n";
    # print data for port1
    print PATTERN "$b\n";
    print D1 "$bdata\n";
}
    }
}
close PATTERN;
close D0;
close D1;

# Convert the file to hexdecimal
open(PATTERN, " $pattern");
open(HEX, "> $hex");
open(P, "> $ports");
print HEX "FF FF FF FF\n" x 3 . "FF\n";
$e = 0;
while(<PATTERN>){
    $i = 0;
    chop;
    for($off = 0; $off < length; $off = $off+4){
$s = substr($_, $off, 4);
$d[$i++] = &bin2hex($s);
    }
    $e = $e % 2;
    if($e == 0){
print P "$d[0] $d[1]$d[2]$d[3]$d[4] ";
    }else{
print P "$d[0] $d[1]$d[2]$d[3]$d[4]\n";
    }
    $e++;
    print HEX "$d[0]$d[1] $d[2]$d[3] $d[4]1 FF\n";
```

```
}

# This subroutine converts hexdecimal to 16-bit binary.
sub hex2bin{
    local($item) = pop(@_);
    local($off, $l) = 0;
    local($c, $bin) = '';
    $l = length($item);
    while($off < $l){
$c = substr($item, $off++, 1);
if($c eq '0'){
    $bin = $bin . '0000';
}elsif($c eq '1'){
    $bin = $bin . '0001';
}elsif($c eq '2'){
    $bin = $bin . '0010';
}elsif($c eq '3'){
    $bin = $bin . '0011';
}elsif($c eq '4'){
    $bin = $bin . '0100';
}elsif($c eq '5'){
    $bin = $bin . '0101';
}elsif($c eq '6'){
    $bin = $bin . '0110';
}elsif($c eq '7'){
    $bin = $bin . '0111';
}elsif($c eq '8'){
    $bin = $bin . '1000';
}elsif($c eq '9'){
    $bin = $bin . '1001';
}elsif($c eq 'A'){
    $bin = $bin . '1010';
}elsif($c eq 'B'){
    $bin = $bin . '1011';
}elsif($c eq 'C'){
    $bin = $bin . '1100';
}elsif($c eq 'D'){
    $bin = $bin . '1101';
}elsif($c eq 'E'){
    $bin = $bin . '1110';
}elsif($c eq 'F'){
```

```
    $bin = $bin . '1111';
}
    }
    $l = length($bin);
    if($l > 16){
die "Integer over range\n";
    }elsif($l < 16){
$l = 16 - $l;
$bin = "0" x $l . $bin;
    }
    $bin;
}

# This subroutine converts the binary to hexdecimal.
sub bin2hex{
  local($bin) = pop(@_);
  if ($bin eq "0000"){
     0;
  }elsif ($bin eq "0001"){
     1;
  }elsif ($bin eq "0010"){
     2;
  }elsif ($bin eq "0011"){
     3;
  }elsif ($bin eq "0100"){
     4;
  }elsif ($bin eq "0101"){
     5;
  }elsif ($bin eq "0110"){
     6;
  }elsif ($bin eq "0111"){
     7;
  }elsif ($bin eq "1000"){
     8;
  }elsif ($bin eq "1001"){
     9;
  }elsif ($bin eq "1010"){
     A;
  }elsif ($bin eq "1011"){
     B;
  }elsif ($bin eq "1100"){
```

```
     C;
  }elsif ($bin eq "1101"){
     D;
  }elsif ($bin eq "1110"){
     E;
  }elsif ($bin eq "1111"){
     F;
  }
}
```

B.3 The Loader Program

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <bios.h>
#include <dos.h>
#include <conio.h>
#define PRINTER 0  // 0=lpt1, 1=lpt2
#define STROBE  0
#define LPT 0x378  /* LPT1=0x378, LPT2=0x3bc */
//#define DEBUG
void main( int argc, char **argv)
{
  int idx, i, rc, k, j, l, tmp, g, length = 0;
  unsigned long int clks;
  unsigned int data[10000];
  long stop, bcnt;
  char path[128], buf;
  FILE *f,*c, *hex_id;
  int skip = 0;

  printf ("Initializing. . . \n");
  _bios_printer(_PRINTER_INIT, PRINTER, 0);  //initialize printer
  delay (200);
  printf("\nColt-Eval configurator, version 0.1, 1/97\n\n");

//  for(i=0;i<argc; i++)printf(">%s\n",argv[i]);
  if (argc != 4 ) {
exit(1);
  } else {
 strcpy(path,argv[1]);
```

```
  }
  f = fopen( path, "rb");
  if (f == 0) {
 printf("\nCannot open %s.  Bye.\n\n",path);
  }
  bcnt = 0;

  printf ("\n\n");

//  fseek (f,0,SEEK_END);
//  stop = ftell(f);
//  rewind(f);

  do {
 fread( &buf, 1, 1, f );
 if (skip < 66) {
skip++;
continue;
 }
#ifdef DEBUG
 tmp = buf;
 tmp &= 0x00FF;
 printf (" (%02x)",tmp);
#endif
 if ( feof(f)==0 ) {
 for (i=0; i<8; i++) {
if ((buf & (1 <<(7-i))) != 0) {
outportb( LPT, 0xff); /* data = 1 */
#ifdef DEBUG
printf ("1");
#endif
/* can't use the following because busy could be high
   _bios_printer( _PRINTER_WRITE, PRINTER, 0xff );    */
} else {
outportb( LPT, 0x0); /* data = 0 */
#ifdef DEBUG
printf ("0");
#endif
/* can't use the following because busy could be high
   _bios_printer( _PRINTER_WRITE, PRINTER, 0 );*/
}
```

```
// for (k=0; k<4; k++);
outportb( LPT+2, 4);
// for (k=0; k<4; k++);
outportb( LPT+2, 5); /* Strobe data */
}
/* printf("%c",buf);*/
bcnt++;
 }
  } while (feof(f) == 0);

  printf("Xilinx download complete. %ld bytes transferred.\n", bcnt);

  printf ("Now clocking the chip. \n");
  for(j=0; j<=1; j++){
outportb( LPT+2, 4);
outportb( LPT+2, 5); /* Strobe data */
  }

  // - - - Begin transmission of Colt data stream - - -

  // Convert command line arguments
  hex_id = fopen( argv[2], "r");
  clks = strtoul( argv[3], NULL, 10);

  // Read hex data file into buffer
  while( fscanf( hex_id, "%x", &tmp) != EOF)
data[length++] = tmp;

  printf("Stream file %s contains %d nibbles.\n", argv[2], length);
  printf("Clocking %d words of the stream to board.\n", clks);

  // Dump buffer out the parallel port
  // - The first nine values in buffer are FF to allow for reset
  // - After that, every 4 buffer words represent a single Colt word
  for( idx = 0; idx <= 4 * clks + 8; idx++) {

// When end of buffer is reached, send FF
if( idx > length) tmp = 0xff;
else tmp = data[idx];

// Send data word on lower byte
```

```
outportb( LPT+0, tmp);

// Strobe clock lines on high byte
outportb( LPT+2, 4);
outportb( LPT+2, 5);
  }
}
```

B.4 VHDL code of the Stream Controller

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.finc.all;

entity MX is
  port(CLK: in std_logic ;
       PC: in std_logic_vector(7 downto 0);
       D: out std_logic_vector(7 downto 0);
       A: inout std_logic_vector(3 downto 0);
       C1_P0, C2_P0, C3_P5: inout std_logic_vector(19 downto 0);
       C1_P1: inout std_logic_vector(19 downto 0);
       RD_B, LDC, RESET: out std_logic;
       COLT_CLK : out std_logic;
       COLT_CLK_B : out std_logic;
       WR_B : inout std_logic);
end MX;

architecture ALG of MX is
  signal SG: std_logic_vector(8 downto 0):="000000001";
  signal COUNT : std_logic_vector(3 downto 0):="0000";
  signal STATE: std_logic_vector(3 downto 0):="0000";
  signal D0, D1 : std_logic_vector(19 downto 0);
  signal TMP_CLK, CLKA, S0: std_logic;

begin
  RD_B <= '1';
  WR_B <= CLK;
  LDC <= S0 ;
  COLT_CLK <= CLK when S0 = '1' else  -- the clock of Colt
              TMP_CLK ;
  S0 <= not (STATE(0) or STATE(1) or STATE(2) or STATE(3));
```

```
process(CLK)
begin
  if CLK'event and CLK = '1' then
    SG <= SG(7 downto 0) & SG(8); -- a shift register
    case STATE is
    -- STATE 0001 => get 1st byte of port0;
    -- STATE 0010 => get 2nd byte of port0;
    -- STATE 0011 => get 3rd byte of port0;
    -- STATE 1001 => get 1st byte of port1;
    -- STATE 1010 => get 2nd byte of port1;
    -- STATE 1011 => get 3rd byte of port1;
      when "0000" => -- "0000" intialize the LCD and Colt
                  if SG(8) = '1' then
                    RESET <= '1';
                    STATE <= "0001";
                    TMP_CLK <= '0';
                  else
                    RESET <= '0';
                    C1_P0 <= "11011111111111111111";
                    C1_P1 <= "11011111111111111111";
                  end if;
      when "0001" =>
      -- 16 => RX
      -- 17 => TX
      -- 18 => RW
      -- 19 => PROG
   D0(19 downto 12) <= PC;
                  STATE <= "0010";
      when "0010"  =>
                  D0(11 downto 4) <= PC;
                  STATE <= "0011";
      when "0011" =>
                  D0(3 downto 0) <= PC(7 downto 4);
                  STATE <= "0100";
                  TMP_CLK <= '1'; -- The rising edge of the clock
      when "0100" =>
                  STATE <= "1001";
      when "1001" =>
      -- 16 => RX
      -- 17 => TX
      -- 18 => RW
```

```
        -- 19 => PROG
     D1(19 downto 12) <= PC;
                     STATE <= "1010";
        when "1010"  =>
                     D1(11 downto 4) <= PC;
                     STATE <= "1011";
        when "1011" =>
                     D1(3 downto 0) <= PC(7 downto 4);
                     STATE <= "1100";
                     TMP_CLK <= '0'; -- The falling edge of the clock
        when "1100" => -- route the signal to the specified port
                     C1_P0 <= D0;
                     C1_P1 <= D1;
                     STATE <= "0001";
        when others => null;
        end case;
      end if;
    end process;
end ALG;
```

B.5 Tier1 Program of Matrix Multiplication

```
/* a configuration of Colt for the matrix multiplication */
/* author: Tsung-Han Yang   */

#define HIPORT 1
#define LOPORT 2

/* This macro sets the output of ALU equal to (leftreg - 1).
   It is necessary to set carry to 1 in order to use this macro.
   When the carry out of ALU changed, it is negative. */
macro operator decleft ()
  P = 3;
  G = 0;
  R = 9;
end macro;

ports
  a = 2;
  b = 1;
  outdata = 6;
end ports;
```

```
stream ma(in a)
  port a => mult at LOPORT;
  mult at LOPORT => skip at 1;
end stream;

stream mb(in b, out outdata)
  port b => mult at HIPORT;
  mult at HIPORT => ifu const at 1;

  block const
    skip data
      south from opposite;
    end skip;
  end, go south to count;

  // Block to count the number of accumulation
  block count
    rightreg = from skip north;
    leftreg = loop validdata(0);
    out = ifcond
            decleft
          else right;
    cond != condout;
    condout = carryout;
    valid = leftandright16;
    carry != zero;
    skip data
      south from opposite;
    end skip;
    skip cond
      south from condout;
    end skip;
  end, go south to accum;

  // Block to accumulate the outputs of the multiplier
  block accum
    rightreg = from skip north;
    leftreg = loop validzero;
    out = add;
    valid = leftandright16;
```

```
    cond != from skip north;
    skip data
      south from out;
    end skip;
    skip cond
      south from opposite;
    end skip;
  end, go south to hout;

  // Block to store the output
  block hout
    leftreg = from local north;
    out = passleft;
    cond = from skip north;
    valid = cond;
  end, go to crossbar;

  ifu hout => port outdata;
end stream;
```

VITA

Tsung-Han Yang was born on March 29, 1969, in Kaohsiung, Taiwan. He received a Bachelor of Science Degree in Electrical Engineering from National Cheng-Kung University in June, 1991. In August, 1995, he enrolled in the graduate program at Virginia Tech to pursue a Master of Science in Electrical Engineering. His focus is on VLSI design and microprocessors. Tsung-Han will be joining Intel in September, 1997, where he will be doing VLSI design.