

**ARITHMETIC UNITS FOR A HIGH PERFORMANCE  
DIGITAL SIGNAL PROCESSOR**

By

MICHAEL ANDREW LAI  
B.S. (University of California, Davis) 2002

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

---

Chair, Dr. Bevan Baas

---

Member, Dr. Rajeevan Amirtharajah

---

Member, Dr. G. Robert Redinbo

Committee in charge  
2004

© Copyright by Michael Andrew Lai 2004  
All Rights Reserved

# Abstract

An Arithmetic Logic Unit (ALU) and a Multiply-Accumulate (MAC) unit designed for a high performance digital signal processor are presented. The 16-bit ALU performs all the logic instructions and includes a saturating adder and subtractor. It also performs shift instructions and a bit reverse instruction. The MAC unit is pipelined into three stages and includes a 40-bit accumulator. It utilizes Modified Booth's algorithm, reduces the partial product tree with rows of 4:2 compressors and half adders, and produces the final result with a 40-bit carry select adder. The MAC unit also contains a 40-bit right shifter for the accumulator. Both units are laid out and tested in the TSMC 0.18  $\mu\text{m}$  process. The ALU can be clocked at 398 MHz and the MAC unit can be clocked at 611 MHz at a supply voltage of 1.8 V and a temperature of 40°C.

# Acknowledgments

First and foremost, I thank my advisor, Professor Bevan Baas. Your guidance has not only taught me a great deal about technical matters, but also to be a more confident person. Your patience and optimism are qualities that make a great advisor. You gave me a chance when it seemed like no one else would. I will always be grateful for that.

To Professor Rajeevan Amirtharajah and Professor G. Robert Redinbo, I thank you for being readers on my committee. I appreciated your comments, direction, and feedback.

To Intel Corporation, thank you for the generous donation of computers on which most of this work was done.

To mom, your love and support have kept me going all these years. You are the strongest woman I have ever known, and I am still amazed every day how you were able to raise Marissa and I. Thank you for always being there for me. I love you.

To mui, you have grown into a beautiful woman, and although I may never tell you, I am very proud of you. Thank you for always cheering me up, and for looking out for me even though I am your older brother.

To dad, thank you for your encouraging words, and always telling me to stay focused. I may not have any engineering degrees were it not for your support. My strong desire to always improve myself and become successful comes from your teachings. I will always be a student of yours.

To Omar, Ryan, Mike, and Zhiyi, thank you for making my stay as a graduate student so fun. You were all willing to help me whenever I asked, and your personalities have taught me a lot about how different people can be, yet still get along. I will miss the fast food lunches, funny faces, and pocket rockets. These days, I will ride in blue thunder while playing golf and eating cheap yogurt.

To Leah, your continuing support throughout graduate school has taught me how selfless people can be. You have always been there for me with your encouragement and advice. Because of you, I am a happier person.

There are many others that have helped me achieve this accomplishment. I thank my friends and I especially thank Karin Mack, Diana Keen, and Senthilkumar Cheetancheri for their mentorship and for introducing me to the benefits of graduate school. You first helped me pave the road to this thesis.

# Contents

|   |             |
|---|-------------|
| <b>Abstract</b>   | <b>ii</b>   |
| <b>Acknowledgments</b>  | <b>iii</b>  |
| <b>List of Figures</b>  | <b>vi</b>   |
| <b>List of Tables</b>   | <b>viii</b> |
| <b>1 Introduction</b>   | <b>1</b>    |
| 1.1 Project Goals . . . . .                                     | 1           |
| 1.1.1 Target System . . . . .                                   | 2           |
| 1.2 Overview . . . . .  | 2           |
| <b>2 Adder Algorithms and Implementations</b>                   | <b>4</b>    |
| 2.1 Basic Adder blocks . . . . .                                | 4           |
| 2.1.1 Half Adder . . . . .                                      | 4           |
| 2.1.2 Full Adder . . . . .                                      | 5           |
| 2.1.3 Partial Full Adder . . . . .                              | 6           |
| 2.2 Adder Algorithms . . . . .                                  | 8           |
| 2.2.1 Ripple Carry Adder . . . . .                              | 8           |
| 2.2.2 Carry Skip Adder . . . . .                                | 9           |
| 2.2.3 Carry Look Ahead Adder . . . . .                          | 12          |
| 2.2.4 Carry Select Adder . . . . .                              | 18          |
| 2.3 Algorithm Analysis . . . . .                                | 20          |
| 2.4 Implementation of a 16-bit Carry Look Ahead Adder . . . . . | 21          |
| 2.4.1 Partial Full Adders . . . . .                             | 22          |
| 2.4.2 Carry look ahead logic for 4 bits . . . . .               | 23          |
| 2.4.3 4-bit CLA adder . . . . .                                 | 23          |
| 2.4.4 16-bit CLA adder . . . . .                                | 24          |
| 2.4.5 Critical Path Analysis . . . . .                          | 25          |
| 2.4.6 Layout and Performance . . . . .                          | 27          |
| 2.5 Summary . . . . .   | 30          |
| <b>3 Multiplication Schemes</b>                                 | <b>31</b>   |
| 3.1 Multiplication Definition . . . . .                         | 31          |
| 3.2 Array Multiplier . . . . .                                  | 33          |
| 3.3 Tree Multiplier . . . . .                                   | 35          |
| 3.3.1 Wallace Tree . . . . .                                    | 35          |
| 3.3.2 Dadda Tree . . . . .                                      | 36          |
| 3.3.3 4:2 Carry Save Compressor . . . . .                       | 37          |
| 3.4 Partial Product Generation Methods . . . . .                | 39          |

|          |   |           |
|----------|---|-----------|
| 3.4.1    | Booth's Algorithm . . . . .                                 | 39        |
| 3.4.2    | Modified Booth's Algorithm . . . . .                        | 40        |
| 3.5      | Summary . . . . .   | 42        |
| <b>4</b> | <b>Arithmetic Logic Unit Design and Implementation</b>      | <b>43</b> |
| 4.1      | Instruction Set . . . . .                                   | 43        |
| 4.2      | Instruction Set Design and Implementation . . . . .         | 45        |
| 4.2.1    | Logic Operations Design and Implementation . . . . .        | 45        |
| 4.2.2    | Word Operations Design and Implementation . . . . .         | 45        |
| 4.2.3    | Bit Reverse and Shifter Design and Implementation . . . . . | 47        |
| 4.2.4    | Adder/Subtractor Design and Implementation . . . . .        | 52        |
| 4.2.5    | Operational Code Selection . . . . .                        | 55        |
| 4.2.6    | Decode Logic . . . . .                                      | 55        |
| 4.3      | Layout and Performance . . . . .                            | 59        |
| 4.4      | Summary . . . . .   | 63        |
| <b>5</b> | <b>Multiply-Accumulate Unit Design and Implementation</b>   | <b>65</b> |
| 5.1      | Instruction Set . . . . .                                   | 65        |
| 5.2      | Instruction Set Design and Implementation . . . . .         | 65        |
| 5.2.1    | Partial Product Generation . . . . .                        | 66        |
| 5.2.2    | Partial Product Accumulation . . . . .                      | 73        |
| 5.2.3    | Final stage for the MAC unit . . . . .                      | 77        |
| 5.2.4    | Decode Logic . . . . .                                      | 81        |
| 5.3      | Layout and Performance . . . . .                            | 82        |
| 5.4      | Summary . . . . .   | 86        |
| <b>6</b> | <b>Conclusion</b>   | <b>87</b> |
| 6.1      | Summary . . . . .   | 87        |
| 6.2      | Future Work . . . . .                                       | 87        |
|          | <b>Bibliography</b>   | <b>89</b> |

# List of Figures

|      |  |    |
|------|--|----|
| 2.1  | Gate Schematic for a Half Adder . . . . .  | 5  |
| 2.2  | Gate Schematic for a Full Adder . . . . .  | 6  |
| 2.3  | Gate Schematic for a Partial Full Adder (PFA) . . . . .  | 7  |
| 2.4  | Schematic for an N-bit Ripple Carry Adder . . . . .  | 8  |
| 2.5  | Critical Path for an N-bit Ripple Carry Adder . . . . .  | 8  |
| 2.6  | One group in a Carry Skip Adder. In this case $M=4$ . . . . .                                  | 10 |
| 2.7  | A 16 bit Carry Skip Adder. $N=16, M=4$ in this figure. . . . .                                 | 10 |
| 2.8  | Critical Path through a 16-bit CSKA . . . . .  | 11 |
| 2.9  | 4-bit carry look ahead adder . . . . .   | 14 |
| 2.10 | Gate Schematic for 4-bit carry look ahead logic in a NAND-NAND network . . . . .               | 15 |
| 2.11 | Gate Schematic for <i>group generate</i> and <i>propagate</i> in a NAND-NAND network . . . . . | 16 |
| 2.12 | Schematic for a 16-bit CLA adder . . . . .   | 17 |
| 2.13 | Critical Path for a 16-bit CLA adder . . . . .   | 17 |
| 2.14 | Schematic for a 16-bit CLSA with 8-bit RCA blocks . . . . .                                    | 18 |
| 2.15 | Schematic for a 16-bit CLSA with 4-bit RCA blocks . . . . .                                    | 19 |
| 2.16 | Schematic for XOR and XNOR gates . . . . .   | 23 |
| 2.17 | Gate Schematic for the implemented PFA . . . . .   | 23 |
| 2.18 | Block Diagram for the implemented 4-bit CLA adder . . . . .                                    | 24 |
| 2.19 | Two XOR gates for sign extension . . . . .   | 25 |
| 2.20 | A 16-bit signed CLA adder . . . . .  | 25 |
| 2.21 | Schematic of transitions for the 16-bit CLA adder . . . . .                                    | 26 |
| 2.22 | Energy-delay plot for the 16-bit signed adder . . . . .  | 29 |
| 2.23 | Final layout for the 16-bit signed adder . . . . .   | 29 |
|      |  |    |
| 3.1  | Generic Multiplier Block Diagram . . . . .   | 32 |
| 3.2  | Partial product array for an $M \times N$ multiplier . . . . .                                 | 33 |
| 3.3  | Partial product array for a 4x4 multiplier . . . . .   | 34 |
| 3.4  | Array multiplier block diagram for a 4x4 multiplier . . . . .                                  | 34 |
| 3.5  | Dot Diagrams for a half adder and a full adder . . . . .                                       | 34 |
| 3.6  | Wallace Tree for an $8 \times 8$ -bit partial product tree . . . . .                           | 35 |
| 3.7  | Dadda Tree for an $8 \times 8$ -bit partial product tree . . . . .                             | 36 |
| 3.8  | 4:2 compressor block diagram . . . . .   | 37 |
| 3.9  | A chain of 4:2s . . . . .  | 37 |
| 3.10 | A 4:2 compressor made of 2 chained full adders . . . . .                                       | 38 |
| 3.11 | 4:2 compressor tree for an $8 \times 8$ -bit multiplication . . . . .                          | 38 |
| 3.12 | Recoded multiplier using Modified Booth Encoding . . . . .                                     | 40 |
| 3.13 | Bewick's implementation of the Booth encoder and decoder . . . . .                             | 41 |
|      |  |    |
| 4.1  | Array of 16 bit-slices for the ALU . . . . .   | 46 |
| 4.2  | 4-to-1 selector using pass transmission gates . . . . .  | 46 |
| 4.3  | ANDWORD,ORWORD, and XORWORD gate schematics . . . . .  | 47 |

|      |   |    |
|------|---|----|
| 4.4  | 16-bit logical left barrel shifter . . . . .                                    | 48 |
| 4.5  | 16-bit logical and arithmetic right barrel shifter . . . . .                    | 49 |
| 4.6  | Illustration of fully utilized horizontal tracks in left shifter . . . . .      | 50 |
| 4.7  | Block Diagram for left and right shifter, and bit reverse instruction . . . . . | 51 |
| 4.8  | Adder/subtractor block diagram . . . . .  | 54 |
| 4.9  | Decode Logic and ALU block diagram, in separate pipe stages . . . . .           | 58 |
| 4.10 | Gate Schematic for the Decode Logic for the ALU . . . . .                       | 60 |
| 4.11 | Block Diagram for ALU . . . . .   | 61 |
| 4.12 | Final layout for the ALU . . . . .  | 62 |
| 4.13 | Schematic for negative edge triggered D flip-flop used in ALU . . . . .         | 63 |
|      |   |    |
| 5.1  | Block diagram for MAC unit . . . . .  | 67 |
| 5.2  | Partial Product Tree with sign extension . . . . .                              | 68 |
| 5.3  | Reducing sign extension in a partial product tree . . . . .                     | 70 |
| 5.4  | Block diagram for Booth encoder and partial product selector . . . . .          | 71 |
| 5.5  | Gate schematic for Booth encoder . . . . .                                      | 71 |
| 5.6  | Gate schematic for a partial product bit selector . . . . .                     | 72 |
| 5.7  | Gate schematic for the partial product row selector . . . . .                   | 72 |
| 5.8  | Partial Product tree reduction with 4:2 compressors and half adders . . . . .   | 74 |
| 5.9  | 4:2 compressor gate schematic . . . . .   | 75 |
| 5.10 | 4:2 compressor transistor implementation . . . . .                              | 75 |
| 5.11 | Half adder transistor implementation . . . . .                                  | 77 |
| 5.12 | Dot diagram of bits in the third pipe stage . . . . .                           | 78 |
| 5.13 | Full adder transistor implementation . . . . .                                  | 79 |
| 5.14 | 24-bit carry propagate adder . . . . .  | 80 |
| 5.15 | 40-bit carry select . . . . .   | 80 |
| 5.16 | Final layout for the MAC unit . . . . .   | 85 |

# List of Tables

|      |   |    |
|------|---|----|
| 2.1  | Truth table for a Half Adder . . . . .  | 5  |
| 2.2  | Truth table for a Full Adder . . . . .  | 6  |
| 2.3  | Extended Truth Table for a 1-bit adder . . . . .  | 7  |
| 2.4  | Transistor count for 8-bit RCA and CSKA adders . . . . .  | 20 |
| 2.5  | Transistor Count for 8-bit CLA adder and CSLA adders . . . . .  | 20 |
| 2.6  | Adder Comparison . . . . .  | 21 |
| 2.7  | Number of signals driven by each <i>propagate</i> and <i>generate</i> signal in the 4-bit CLA logic . . . . . | 22 |
| 2.8  | Input Signals and Transitions for the 16-bit CLA adder . . . . .  | 27 |
| 2.9  | 16-bit CLA adder results . . . . .  | 28 |
| 2.10 | Energy-delay product for the 16-bit signed adder . . . . .  | 29 |
|      |   |    |
| 3.1  | Wallace and Dadda Comparison for an $8 \times 8$ -bit partial product tree . . . . .                          | 36 |
| 3.2  | Wallace, Dadda, and 4:2 Comparison for an $8 \times 8$ -bit multiplier . . . . .                              | 38 |
| 3.3  | Original Booth Algorithm . . . . .  | 40 |
| 3.4  | Modified Booth Algorithm . . . . .  | 41 |
|      |   |    |
| 4.1  | Instruction set for the ALU in AsAP . . . . .   | 44 |
| 4.2  | Truth Table for XOR, used as a programmable inverter . . . . .  | 45 |
| 4.3  | Area comparison for Bit Reverse and Shifter Implementation . . . . .  | 51 |
| 4.4  | Conditions for saturation and results . . . . .   | 53 |
| 4.5  | Add/Sub arithmetic equations and details . . . . .  | 54 |
| 4.6  | Addition and subtraction opcodes and control signals . . . . .  | 56 |
| 4.7  | Logic, Word, and Shift opcodes and control signals . . . . .  | 57 |
| 4.8  | ALU performance . . . . .   | 64 |
|      |   |    |
| 5.1  | Instruction Set for the MAC unit in AsAP . . . . .  | 66 |
| 5.2  | Truth table for the Booth encoder . . . . .   | 69 |
| 5.3  | Truth table for the partial product bit selector . . . . .  | 73 |
| 5.4  | Truth table for MAC control signals . . . . .   | 82 |
| 5.5  | MAC unit performance . . . . .  | 83 |

# Chapter 1

## Introduction

Digital Signal Processing (DSP) is finding its way into more applications [1], and its popularity has materialized into a number of commercial processors [2]. Digital signal processors have different architectures and features than general purpose processors, and the performance gains of these features largely determine the performance of the whole processor. The demand for these special features stems from algorithms that require intensive computation, and the hardware is often designed to map to these algorithms. Widely used DSP algorithms include the Finite Impulse Response (FIR) filter, Infinite Impulse Response (IIR) filter, and Fast Fourier Transform (FFT). Efficient computation of these algorithms is a direct result of the efficient design of the underlying hardware.

One of the most important hardware structures in a DSP processor is the Multiply-Accumulate (MAC) unit. This unit can calculate the running sum of products, which is at the heart of algorithms such as the FIR and FFT. The ability to compute with a fast MAC unit is essential to achieve high performance in many DSP algorithms, and is why there is at least one dedicated MAC unit in all of the modern commercial DSP processors [3].

### 1.1 Project Goals

The goal of this project is to design and implement a MAC unit and an Arithmetic Logic Unit (ALU). The MAC unit is a 16x16-bit 2's complement multiplier with a 40-bit accumulator. The ALU performs 16-bit arithmetic and includes saturating addition/subtraction logic. The implementation includes a full custom layout and verification of all cells necessary to complete the units. We perform all our simulations for the TSMC 0.18  $\mu\text{m}$  process, and the chip that uses these units

will be fabricated. The priorities of this project, in order of importance, are:

- 1) Robust and safe circuits.
- 2) Design time
- 3) Area/speed balance

The most important priority during this project is to ensure that it works. Thus, the circuits must be robust and safe, and we must choose designs that are largely immune to noise and generate full rail-to-rail swing on the outputs. To be safe, all our circuits are designed with static CMOS, which means at every point in time, each gate output is connected to either  $V_{dd}$  or  $Gnd$  [4]. Secondly, design time in optimizing the performance of circuits is very valuable. We want to spend the most time on issues that will reap the greatest gains. For example, fully optimizing the size of transistors may increase the performance only a fraction of what can be achieved by choosing a better architecture without optimized sizing. And thirdly, we want to choose the right balance between the speed of our circuits and the area. For critical paths in the MAC and ALU, more area will be devoted to speed up the circuit. Another important issue in digital circuits besides speed and area is power consumption. In this work, our main focus is on performance, but we do provide some numbers for the arithmetic units relating to energy and power. This is to provide an estimate of the amount of energy and power consumed by the units we choose to implement.

### 1.1.1 Target System

The work in this thesis is part of a larger project: the Asynchronous Array of simple Processors (AsAP) chip [5]. The AsAP chip is a parallel, programmable, and reconfigurable two-dimensional array of processors. Each processor has a 16-bit fixed point datapath and a 9-stage pipeline. They also each have small, local memories that are used for instructions, data, and configuration values. Each processor contains two First-In First-Out (FIFO) memory structures to facilitate the communication between itself and other processors in the array [6]. The chip is designed for DSP algorithms and workloads, hence the saturating logic in the ALU and the MAC unit in the datapath.

## 1.2 Overview

Chapter 2 begins with an overview of the different types of adders and certain design considerations to build faster adders. It concludes with the design and implementation of a 16-bit signed Carry Look Ahead adder. Chapter 3 discusses the multiplication operation and techniques to build

fast multipliers. Chapter 4 presents the design and implementation of an Arithmetic Logic Unit, which includes shifters and a saturating adder. Chapter 5 describes the design and implementation of a 16-bit 2's complement multiply-accumulate unit with a 40-bit accumulator. Finally, Chapter 6 concludes the thesis by summarizing the work and considers areas of future work.

## Chapter 2

# Adder Algorithms and Implementations

In nearly all digital IC designs today, the addition operation is one of the most essential and frequent operations. Instruction sets for DSP's and general purpose processors include at least one type of addition. Other instructions such as subtraction and multiplication employ addition in their operations, and their underlying hardware is similar if not identical to addition hardware. Often, an adder or multiple adders will be in the critical path of the design, hence the performance of a design will be often be limited by the performance of its adders. When looking at other attributes of a chip, such as area or power, the designer will find that the hardware for addition will be a large contributor to these areas. It is therefore beneficial to choose the correct adder to implement in a design because of the many factors it affects in the overall chip. In this chapter we begin with the basic building blocks used for addition, then go through different algorithms and name their advantages and disadvantages. We then discuss the design and implementation of the adder chosen for use in a single processor on the AsAP architecture.

## 2.1 Basic Adder blocks

### 2.1.1 Half Adder

The Half Adder (HA) is the most basic adder. It takes in two bits of the same weight, and creates a *sum* and a *carryout*. Table 2.1 shows the truth table for this adder. If the two inputs  $a$  and  $b$  have a weight of  $2^i$  (where  $i$  is an integer), *sum* has a weight of  $2^i$ , and *carryout* has a weight

| Inputs   |          | Outputs         |            |
|----------|----------|-----------------|------------|
| <i>a</i> | <i>b</i> | <i>carryout</i> | <i>sum</i> |
| 0        | 0        | 0               | 0          |
| 0        | 1        | 0               | 1          |
| 1        | 0        | 0               | 1          |
| 1        | 1        | 1               | 0          |

Table 2.1: Truth table for a Half Adder

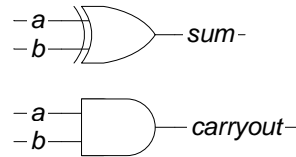


Figure 2.1: Gate Schematic for a Half Adder

of  $2^{(i+1)}$ . Equations 2.1 and 2.2 are the Boolean equations for *sum* and *carryout*, respectively. Figure 2.1 shows a possible implementation using logic gates to realize the half adder.

$$sum = a \oplus b \quad (2.1)$$

$$carryout = a \cdot b \quad (2.2)$$

### 2.1.2 Full Adder

The Full Adder (FA) is useful for additions that have multiple bits in each of its operands. It takes in three inputs and creates two outputs, a *sum* and a *carryout*. The inputs have the same weight,  $2^i$ , the *sum* output has a weight of  $2^i$ , and the *carryout* output has a weight of  $2^{(i+1)}$ . The truth table for the FA is shown in Table 2.2. The FA differs from the HA in that it has a *carryin* as one of its inputs, allowing for the cascading of this structure which is explored below in Section 2.2.1. Equations 2.3 and 2.4 are the Boolean equations for the FA *sum* and FA *carryout*, respectively. In both those equations *cin* means *carryin*. Figure 2.2 shows a possible implementation using logic gates to realize the full adder.

$$sum_i = a_i \oplus b_i \oplus cin_i \quad (2.3)$$

$$carryout_{i+1} = a_i \cdot b_i + b_i \cdot cin_i + a_i \cdot cin_i = a_i \cdot b_i + (a_i + b_i) \cdot cin_i \quad (2.4)$$

| Inputs         |          |          | Outputs         |            |
|----------------|----------|----------|-----------------|------------|
| <i>carryin</i> | <i>a</i> | <i>b</i> | <i>carryout</i> | <i>sum</i> |
| 0              | 0        | 0        | 0               | 0          |
| 0              | 0        | 1        | 0               | 1          |
| 0              | 1        | 0        | 0               | 1          |
| 0              | 1        | 1        | 1               | 0          |
| 1              | 0        | 0        | 0               | 1          |
| 1              | 0        | 1        | 1               | 0          |
| 1              | 1        | 0        | 1               | 0          |
| 1              | 1        | 1        | 1               | 1          |

Table 2.2: Truth table for a Full Adder

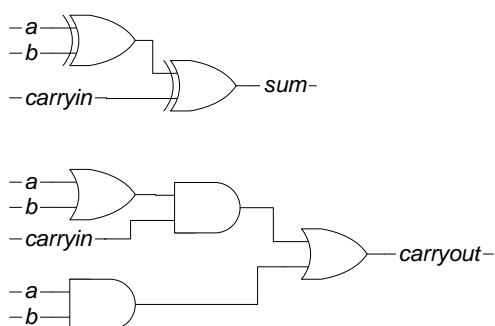


Figure 2.2: Gate Schematic for a Full Adder

### 2.1.3 Partial Full Adder

The Partial Full Adder (PFA) is a structure that implements intermediate signals that can be used in the calculation of the carry bit. Revisiting the truth table for a FA (Table 2.2), we extend it to include the signals *generate* (*g*), *delete* (*d*), and *propagate* (*p*). When  $g=1$ , it means *carryout* will be 1 (generated) regardless of *carryin*. When  $d=1$ , it means *carryout* will be 0 (deleted) regardless of *carryin*. When  $p=1$ , it means *carryout* will equal *carryin* (*carryin* will be propagated). Table 2.3 reflects these three additional signals, with a comment on the *carryout* bit in an additional column. Equations 2.5 – 2.7 are the Boolean equations for *generate*, *delete*, and *propagate*, respectively. It should be noted that for the *propagate* signal, the XOR function can also be used, since in the case of  $a,b=1$ , the *generate* signal will assert that *carryout* is 1. The Boolean equations for the sum and carryout can now be written as functions of *g*, *p*, or *d*. Equations 2.8 and 2.9 show *sum* and *carryout* as functions of *g* and *p* (for Equation 2.8, *p* must be implemented with the XOR). Figure 2.3 shows a circuit for creating the *generate*, *propagate*, and *sum* signals. It is a *partial* full adder because it does not calculate the *carryout* signal directly; rather, it creates the

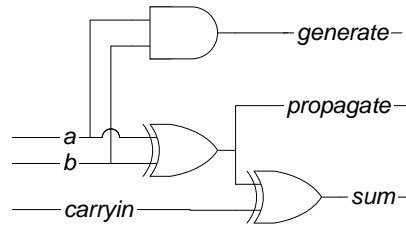


Figure 2.3: Gate Schematic for a Partial Full Adder (PFA)

signals needed to calculate the *carryout* signal. This will be useful for an adder scheme presented in Section 2.2.3. The adder in that section uses the PFA as a subcell, and does not require the use of the *delete* signal. Also,  $g, p=0$  imply  $carryout=0$ , which is equivalent to  $d=1$ . Notice in Figure 2.3 that the *propagate* signal is calculated in the same path that calculates the *sum* signal. If we simply take the *propagate* signal from this internal node, it saves the area required to build this structure. Also note that the HA introduced in Section 2.1.1 is the exact hardware needed to implement the  $p$  and  $g$  signals.

| Inputs         |          |          | Outputs         |            |          |          |          | Carry status       |
|----------------|----------|----------|-----------------|------------|----------|----------|----------|--------------------|
| <i>carryin</i> | <i>a</i> | <i>b</i> | <i>carryout</i> | <i>sum</i> | <i>g</i> | <i>d</i> | <i>p</i> |                    |
| 0              | 0        | 0        | 0               | 0          | 0        | 1        | 0        | delete             |
| 0              | 0        | 1        | 0               | 1          | 0        | 0        | 1        | propagate          |
| 0              | 1        | 0        | 0               | 1          | 0        | 0        | 1        | propagate          |
| 0              | 1        | 1        | 1               | 0          | 1        | 0        | 1        | generate/propagate |
| 1              | 0        | 0        | 0               | 1          | 0        | 1        | 0        | delete             |
| 1              | 0        | 1        | 1               | 0          | 0        | 0        | 1        | propagate          |
| 1              | 1        | 0        | 1               | 0          | 0        | 0        | 1        | propagate          |
| 1              | 1        | 1        | 1               | 1          | 1        | 0        | 1        | generate/propagate |

Table 2.3: Extended Truth Table for a 1-bit adder

$$generate_i(g_i) = a_i \cdot b_i \quad (2.5)$$

$$delete_i(d_i) = \bar{a}_i \cdot \bar{b}_i \quad (2.6)$$

$$propagate_i(p_i) = a_i + b_i \text{ (or } a_i \oplus b_i) \quad (2.7)$$

$$sum_i = p_i \oplus carryin_i \quad (2.8)$$

$$carryout_{i+1} = g_i + p_i \cdot carryin_i \quad (2.9)$$

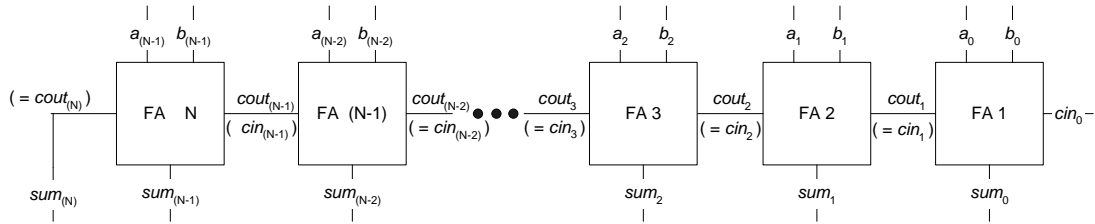


Figure 2.4: Schematic for an N-bit Ripple Carry Adder

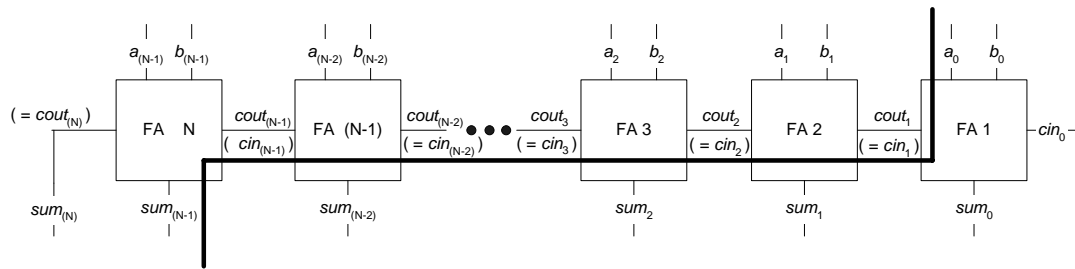


Figure 2.5: Critical Path for an N-bit Ripple Carry Adder

## 2.2 Adder Algorithms

### 2.2.1 Ripple Carry Adder

The *Ripple Carry Adder* (RCA) is one of the simplest adders to implement. This adder takes in two  $N$ -bit inputs (where  $N$  is a positive integer) and produces  $(N + 1)$  output bits (an  $N$ -bit sum and a 1-bit carryout). The RCA is built from  $N$  full adders cascaded together, with the carryout bit of one FA tied to the carryin bit of the next FA. Figure 2.4 shows the schematic for an  $N$ -bit RCA. The input operands are labeled  $a$  and  $b$ , the carryout of each FA is labeled  $cout$  (which is equivalent to the carryin ( $cin$ ) of the subsequent FA), and the sum bits are labeled  $sum$ . Each sum bit requires both input operands and  $cin$  before it can be calculated. To estimate the propagation delay of this adder, we should look at the worst case delay over every possible combination of inputs. This is also known as the *critical path*. The most significant sum bit can only be calculated when the carryout of the previous FA is known. In the worst case (when all the carryouts are 1), this carry bit needs to *ripple* across the structure from the least significant position to the most significant position. Figure 2.5 has a darkened line indicating the critical path.

Hence, the time for this implementation of the adder is expressed in Equation 2.10, where

$t_{RCA_{carry}}$  is the delay for the carryout of a FA and  $t_{RCA_{sum}}$  is the delay for the sum of a FA.

$$\text{Propagation Delay}(t_{RCA_{prop}}) = (N - 1) \cdot t_{RCA_{carry}} + t_{RCA_{sum}} \quad (2.10)$$

From Equation 2.10, we can see that the delay is proportional to the length of the adder. An example of a worst case propagation delay input pattern for a 4 bit ripple carry adder is where the input operands change from 1111 and 0000 to 1111 and 0001, resulting in a sum changing from 01111 to 10000.

From a VLSI design perspective, this is the easiest adder to implement. One just needs to design and lay out one FA cell, and then array  $N$  of these cells to create an  $N$ -bit RCA. The performance of the one FA cell will largely determine the speed of the whole RCA. From the critical path in Equation 2.10, minimizing the carryout delay ( $t_{RCA_{carry}}$ ) of the FA will minimize  $t_{RCA_{prop}}$ . There are various implementations of the FA cell to minimize the carryout delay [4].

### 2.2.2 Carry Skip Adder

From examination of the RCA, the limiting factor for speed in that adder is the propagation of the *cout* bit. The *Carry Skip Adder* (CSKA, also known as the *Carry Bypass Adder*) addresses this issue by looking at groups of bits and determines whether this group has a carryout or not [7]. This is accomplished by creating a group propagate signal ( $p_{CSKA_{group}}$ ) to determine whether the group carryin ( $carryin_{CSKA_{group}}$ ) will propagate across the group to the carryout ( $carryout_{CSKA_{group}}$ ). To explore the operation of the whole CSKA, take an  $N$ -bit adder and divide it into  $N/M$  groups, where  $M$  is the number of bits per group. Each group contains a 2-to-1 multiplexer, logic to calculate  $M$  sum bits, and logic to calculate  $p_{CSKA_{group}}$ . The select line for the mux is simply the  $p_{CSKA_{group}}$  signal, and it chooses between  $carryin_{CSKA_{group}}$  or  $cout_4$ .

To aid the explanation, we refer the reader to Figure 2.6, which shows the hardware for a group of 4 bits ( $M=4$ ) in the CSKA. There are four full adders cascaded together and each FA creates a carryout (*cout*), a propagate (*p*) signal, and a sum (sum not shown). The propagate signal from each FA comes at no extra hardware cost since it is calculated in the sum logic (the hardware is identical to the sum hardware for the PFA shown in Figure 2.3). For the  $carryout_{CSKA_{group}}$  to equal  $carryin_{CSKA_{group}}$ , all of the individual propagates must be asserted (Equations 2.11 and 2.12). If this is true then  $carryin_{CSKA_{group}}$  “skips” past the group of full adders and equals the  $carryout_{CSKA_{group}}$ . For the case where  $p_{CSKA_{group}}$  is 0, at least one of the propagate signals is 0. This implies that either a *delete* and/or *generate* occurred in the group. A *delete* signal simply means that the carryout for the group is 0 regardless of the carryin, and a *generate* signal means



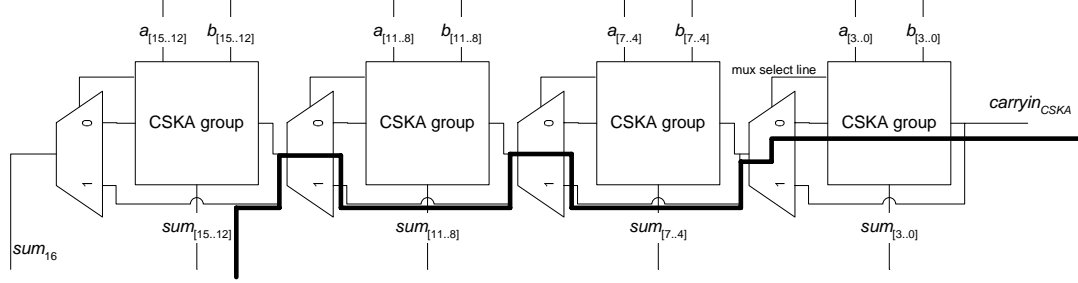


Figure 2.8: Critical Path through a 16-bit CSKA

carryin. In the second case, the carryout signal of the most significant adder will become the group carryout. This means  $p_{CSKAgroup}=0$  and the carryout is independent of the carryin. If we isolate a particular group (as in Figure 2.6), the second case (signal  $cout_4$ ) always takes longer because the carryout signal must be calculated through logic, whereas the first case ( $carryin_{CSKAgroup}$ ) requires only a wire to propagate the signal. Looking at the whole architecture, however, this second case is part of the critical path for only the first CSKA group. Since the second case is not dependent on the group carryin, all the groups in the CSKA can compute the carryout in parallel. If a group needs its carryin ( $p_{CSKAgroup}=1$ ), then it must wait until it arrives after being calculated from a previous group. In the worst case, a carryout must be calculated in the first group, and every group afterwards needs to propagate this carryout. When the final group receives this propagated signal, then it can calculate its sum bits. Figure 2.7 shows a 16-bit CSKA with 4-bit groups and Figure 2.8 shows a darkened line indicating the critical path of the signals in the 16-bit CSKA.

If we assume a 16-bit CSKA with 4-bit groups, with each group containing a 4-bit RCA for the sum logic, then the worst case propagation delay through this adder is expressed in equation 2.13. In this equation,  $t_{RCAcarry}$  and  $t_{RCAsum}$  are the delays to calculate the carryout and sum signals of an RCA, respectively. Each group has 4 bits, so the delay through the first group has 4 RCA carryout delays. This carryout of the first group potentially propagates through 3 muxes, where one mux delay is expressed as  $t_{muxdelay}$ . Finally, when the carryout signal reaches the final group, the sum for this group can be calculated. This is represented by the final two components of Equation 2.13.

$$t_{CSKA_{16}} = 4 * t_{RCAcarry} + 3 * t_{muxdelay} + 3 * t_{RCAcarry} + t_{RCAsum} \quad (2.13)$$

For Equation 2.13, there are some assumptions about the delay through the circuit. First, we assume in the first CSKA group that the group propagate signal is calculated before the carryout of the most significant adder. Thus, the mux for this first group is waiting for the carryout. For the

final CSKA group, we assume that it takes longer for  $sum_{15}$  to be calculated than for  $sum_{16}$  to be calculated. Once the carryin for this last group is known, the delay for  $sum_{16}$  is the delay of the mux; for  $sum_{15}$  it is a delay of  $3 * t_{RCAcarry} + t_{RCAsum}$  (3 ripples through the adder before the last sum bit can be calculated).

For an  $N$ -bit CSKA, the critical path equation is expressed in Equation 2.14.  $M$  represents the number of bits in each group. There are  $\frac{N}{M}$  groups in the adder, and every mux in this group except for the last one is in the critical path. As in Equation 2.13, Equation 2.14 assumes that each group contains a ripple carry adder.

$$t_{CSKA_N} = M * t_{RCAcarry} + \left(\frac{N}{M} - 1\right) * t_{muxdelay} + (M - 1) * t_{RCAcarry} + t_{RCAsum} \quad (2.14)$$

From a VLSI design perspective, this adder shows improved speedup over a RCA without much area increase. The additional hardware comes from the 2-to-1 mux and group propagate logic in each group, which is about 15% more area (based on data presented in Section 2.3). One drawback to this structure is that its delay is still linearly dependent on the width of the adder, therefore for large adders where speed is important, the delay may be unacceptable. Also, there is a long wire in between the groups that  $carryout_{CSKAgroup}$  needs to travel on. This path begins at the carryout of the first CSKA group and ends at the carryin to the final CSKA group. This signal also needs to travel through  $\frac{N}{M} - 1$  muxes, and these will introduce long delays and signal degradation if pass gate muxes are used. If buffers are required in between these groups to reproduce the signal, then the critical path is lengthened. An example of a worst case delay input pattern for a 16-bit CSKA with 4-bit groups is where the input operands are 111111111111000 and 0000000000001000. This forces a carryout in the first group that skips through the middle two groups and enters the final group. This carryin to the final group ripples through to the final sum bit ( $sum_{15}$ ). To determine the optimal speed for this adder, one needs to find the delay through a mux and the carryout delay of a FA. It is one of these two delays that will dominate the delay of the whole CSKA. For short adders ( $\leq 16$  bits), the  $t_{carryout}$  of a FA will probably dominate delay, and for long adders the long wire that skips through stages and muxes will probably dominate the delay.

### 2.2.3 Carry Look Ahead Adder

From the critical path equations in Sections 2.2.1 and 2.2.2, the delay is linearly dependent on  $N$ , the length of the adder. It is also shown in Equations 2.10 and 2.14 that the  $t_{carryout}$  signal contributes largely to the delay. An algorithm that reduces the time to calculate  $t_{carryout}$  and the linear dependency on  $N$  can greatly speed up the addition operation. Equation 2.9 shows that

the carryout can be calculated with  $g$ ,  $p$ , and  $carryin$ . The signals  $g$  and  $p$  are not dependent on  $carryin$ , and can be calculated as soon as the two input operands arrive. Weinberger and Smith invented the *Carry Look Ahead (CLA) Adder* [8]. Using Equation 2.9, we can write the carryout equations for a 4-bit adder. These equations are shown in Equations 2.15 – 2.18, where  $c_i$  represents the carryout of the  $i^{th}$  position ( $0 \leq i \leq (N - 1)$ ), and  $g_i$ ,  $p_i$  represent the generate and propagate signal from each PFA). The equations for  $c_2$ ,  $c_3$  and  $c_4$  are obtained by substitution of  $c_1$ ,  $c_2$  and  $c_3$ , respectively. These equations show that every carryout in the adder can be determined with just the input operands and initial carryin ( $c_3$ ). This process of calculating  $c_i$  by using only the  $p_i$ ,  $g_i$  and  $c_0$  signals can be done indefinitely, however, each subsequent carryout generated in this manner becomes increasingly difficult because of the large number of high fan-in gates [9].

$$c_1 = g_0 + p_0 \cdot c_0 \quad (2.15)$$

$$c_2 = g_1 + p_1 \cdot c_1 = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0 \quad (2.16)$$

$$c_3 = g_2 + p_2 \cdot c_2 = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0 \quad (2.17)$$

$$\begin{aligned} c_4 &= g_3 + p_3 \cdot c_3 \\ &= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0 \end{aligned} \quad (2.18)$$

The CLA adder uses partial full adders as described in Section 2.1.3 to calculate the *generate* and *propagate* signals needed for the carryout equations. Figure 2.9 shows the schematic for a 4-bit CLA adder. The logic for each PFA block is shown in Figure 2.3. The CLA logic block implements the logic in Equations 2.15 – 2.18, and the gate schematic for this block is in Figure 2.10. For a 4-bit CLA adder the 4<sup>th</sup> carryout signal can also be considered as the 5<sup>th</sup> *sum* bit.

Although it is impractical to have a single level of carry look ahead logic for long adders, this can be solved by adding another level of carry look ahead logic. To achieve this, each adder block requires two additional signals: a *group generate* and a *group propagate*. The equations for these two signals, assuming adder block sizes of 4 bits, are shown in Equations 2.19 and 2.20. A *group generate* occurs if a carry is generated in one of adder blocks, and a *group propagate* occurs if the carryin to the adder block will be propagated to the carryout. Figure 2.11 shows the gate schematic of the two additional signals.

$$group\ generate = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot c_3 \quad (2.19)$$

$$group\ propagate = p_0 \cdot p_1 \cdot p_2 \cdot p_3 \quad (2.20)$$

With multiple levels of CLA logic, carry look ahead adders of any length can be built. The size of an adder block in a CLA adder is usually 4 bits because it is a common factor of most word

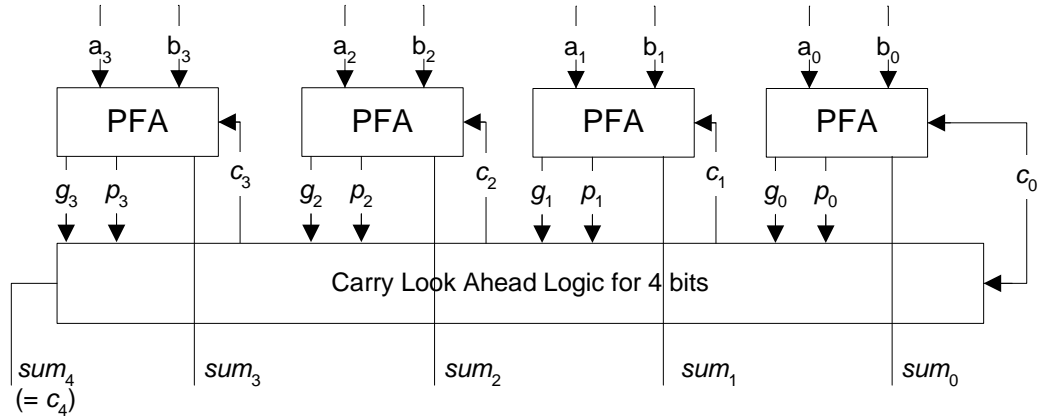


Figure 2.9: 4-bit carry look ahead adder

sizes and there is a practical limit on the gate size that can be implemented [9]. To illustrate the use of another level of CLA logic, Figure 2.12 shows the schematic for a 16-bit CLA adder. There is a second level of CLA logic which takes the *group generate* and *group propagate* signals from each 4-bit adder subcell and calculates the carryout signals for each adder block. If an adder has multiple levels of CLA logic, only the final level needs to generate the  $c_4$  signal. All other levels replace this  $c_4$  signal with the *group generate* and *group propagate*. The CLA logic for this 16-bit adder is identical to the CLA logic for the 4-bit adder in Figure 2.9; therefore the equations for the carryout signals are in Equations 2.15 – 2.18.

A third level of CLA logic and four 16-bit adder blocks can be used to build a 64-bit adder. The CLA logic would create the  $c_{16}$ ,  $c_{32}$ , and  $c_{48}$  signals to be used as carryins to the 16-bit adder blocks and the  $c_{64}$  as the  $sum_{64}$  signal. If a design calls for an adder of length 32, a designer can simply use two 16-bit adder blocks and the first two carryout signals ( $c_{16}$ ,  $c_{32}$ ) from the third level of CLA logic. The identical hardware in the CLA logic, coupled with the fact that the adder blocks can be instantiated as subcells, makes building long adders with this architecture simple.

Determining the critical path for a CLA adder is difficult because the gates in the carry path have different fan-in's. To get a general idea, we first assume that all gate delays are the same. The delay for a 4-bit CLA adder then requires one gate delay to calculate the *propagate* and *generate* signals, two gate delays to calculate *carry* signals, and one gate delay to calculate the *sum* signals; this equates to four gate delays. For a 16-bit CLA adder there is one gate delay to calculate the *propagate* and *generate* signal (from the PFA), two gate delays to calculate the *group propagate* and *generate* in the first level of carry logic, two gate delays for the carryout signals in the second

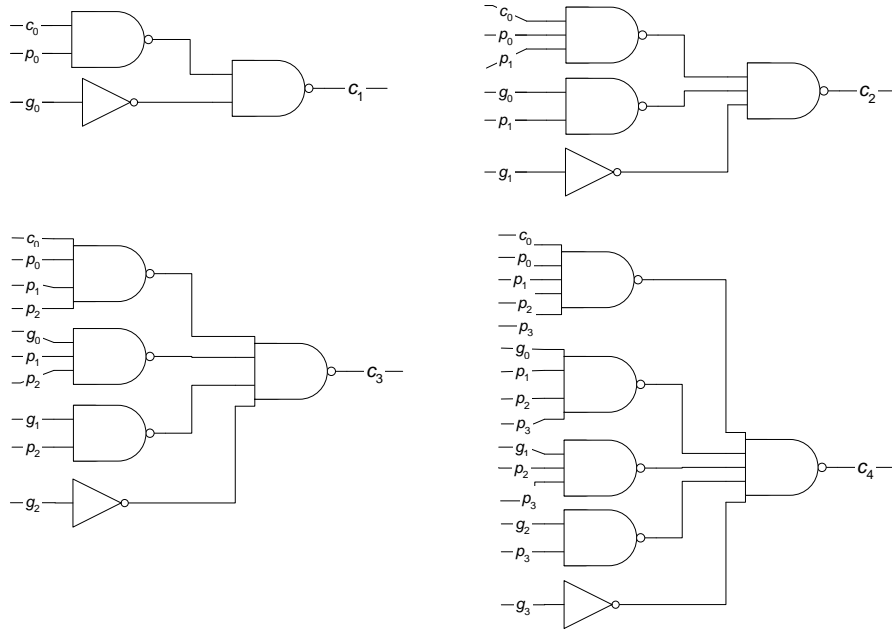


Figure 2.10: Gate Schematic for 4-bit carry look ahead logic in a NAND-NAND network

level of carry logic, and one gate delay for the *sum* signals. The second level of carry logic for the 16-bit CLA adder contributes an additional two gate delays over the 4-bit CLA adder, thus increasing the total to six gate delays. Continuing in this manner (a 64-bit add takes eight gate delays, a 256-bit add takes ten gate delays), we see that the delay for a CLA adder is dependent on the number of levels of carry logic, and not on the length of the adder. If a group size of four is chosen, then the number of levels in an  $N$ -bit CLA is expressed in Equation 2.21 and in general the number of levels in a CLA for a group size of  $k$  is expressed in Equation 2.22. For an  $N$ -bit CLA adder, each level of carry logic introduces two gate delays in addition to a gate delay for the generate and propagate signals and a gate delay for the sum. The total gate delay is expressed in Equation 2.23, which shows that the delay of a CLA adder is logarithmically dependent on the size of the adder. This theoretically results in one of the fastest adder architectures.

$$CLA \text{ levels (with group size of 4)} = \lceil \log_4 N \rceil \tag{2.21}$$

$$CLA \text{ levels (with group size of } k) = \lceil \log_k N \rceil \tag{2.22}$$

$$CLA \text{ gate delay} = 2 + 2 \cdot \lceil \log_k N \rceil \tag{2.23}$$

Equation 2.23, however, lacks the detail necessary to make a good delay estimate. Each gate in the adder varies in both the number of inputs it has and the function it implements. The

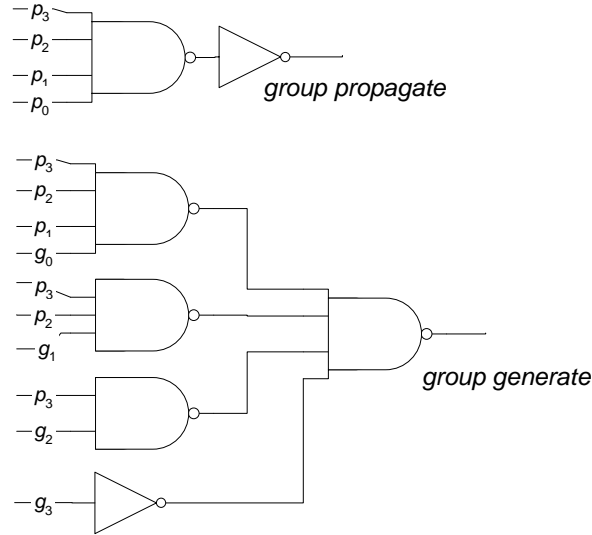


Figure 2.11: Gate Schematic for *group generate* and *propagate* in a NAND-NAND network

slowest gates in the carry logic of Figure 2.10 are the 5-input NAND gates, which when implemented with a single level of logic will contain five NMOS transistors in series. These 5-input NAND gates are in the logic for  $c_4$ , which is the most significant bit for the result of any add. If there are multiple levels of carry logic, the  $c_4$  logic is replaced with the *group propagate* and *generate* signals, and is used only in the final level of carry logic. Also, this signal is not in the critical path because once it is calculated, its result can be immediately used, as opposed to the other carryout signals. Signals  $c_1$ ,  $c_2$ , and  $c_3$  feed into the PFAs, where the *sum* signal still needs to be calculated (another XOR gate delay). The second largest gates are the 4-input NAND gates, with four NMOS transistors in series. These gates are contained in the *group generate* and  $c_3$  logic. The critical path therefore goes through the *group generate* signal (in the first and intermediate levels of carry logic), and the  $c_3$  signal in the last level of carry logic. Figure 2.13 shows a darkened line indicating the critical path of the signals in the 16-bit CLA adder, and Equation 2.24 expresses the critical delay of a 16-bit CLA adder. In this equation,  $t_{prop}$  is the propagate delay for a PFA,  $t_{GroupGen}$  is the delay for the *group generate* signal in the first level of carry logic,  $t_{c_3}$  is the delay for  $c_3$  in the second level of carry logic, and  $t_{xor}$  is the second XOR delay of the PFA to calculate the sum. For an  $N$ -bit CLA adder with 4-bit groups, the delay is expressed in Equation 2.25. The second term in this equation is the number of carry levels (minus 1) multiplied by the delay of the *group generate* signal, and

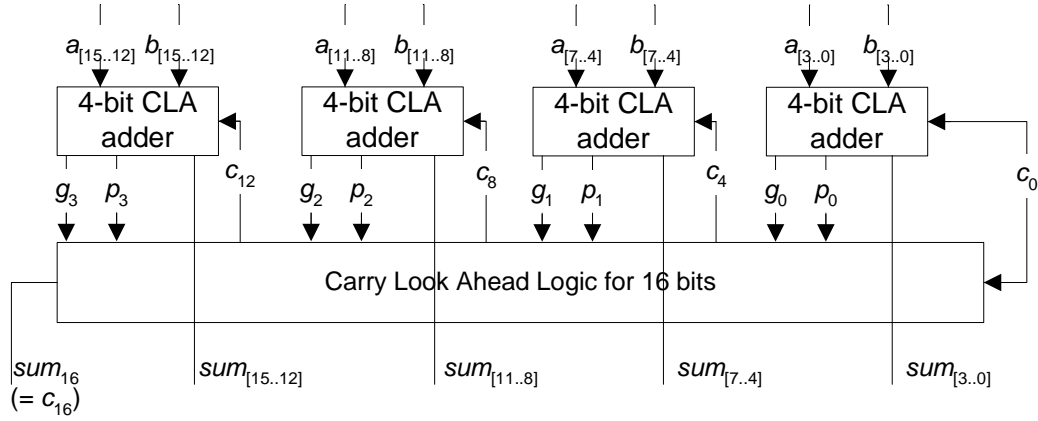


Figure 2.12: Schematic for a 16-bit CLA adder

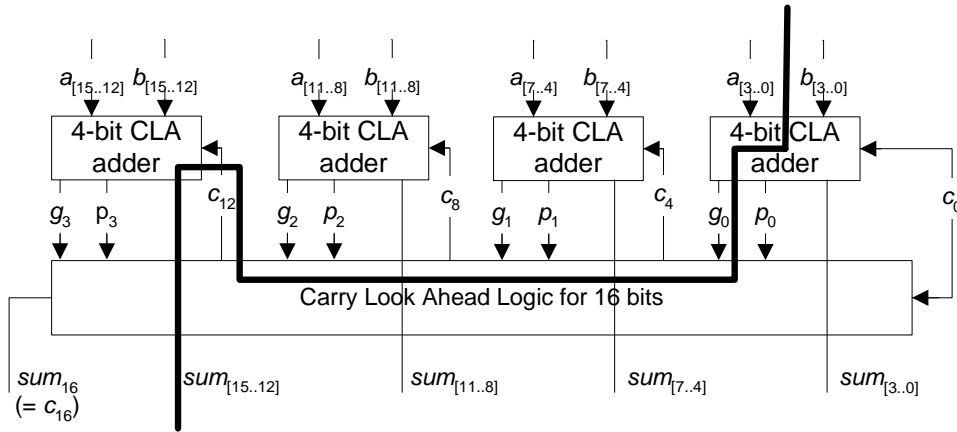


Figure 2.13: Critical Path for a 16-bit CLA adder

shows that the delay is logarithmically dependent on the length of the adder.

$$t_{CLA_{16}} = t_{prop} + t_{GroupGen} + t_{cout_3} + t_{xor} \quad (2.24)$$

$$t_{CLA_N} = t_{prop} + (\lceil \log_4 N \rceil - 1) \cdot t_{GroupGen} + t_{cout_3} + t_{xor} \quad (2.25)$$

From a VLSI design perspective, this adder may take more time to implement, but there still exists a regularity with the architecture that allows building long adders fairly easily. The reuse of the CLA logic definitely contributes to the feasibility of building a long adder without additional design time. Also, after an adder is built, it can be used as a subcell, as is done with the 4-bit adders as blocks in the 16-bit CLA adder. A drawback to CLA adders are their larger areas. There is a large amount of hardware dedicated to calculating the carry bits from cell to cell. However, if the

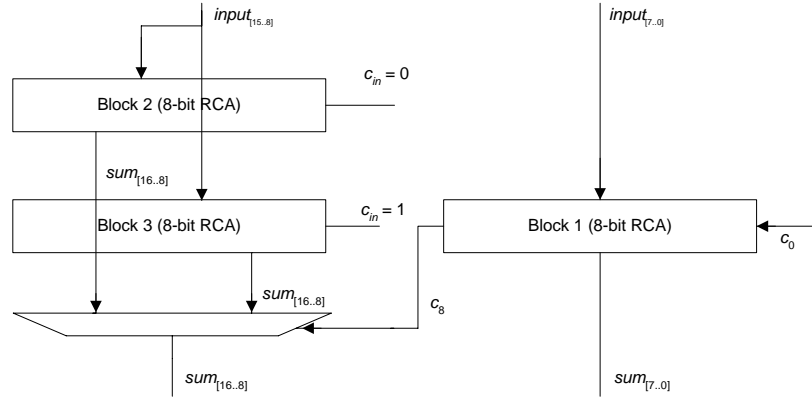


Figure 2.14: Schematic for a 16-bit CLSA with 8-bit RCA blocks

application calls for high performance, then the benefits of decreased delay can outweigh the larger area.

#### 2.2.4 Carry Select Adder

Adding two numbers by using redundancy can speed addition even further. That is, for any number of sum bits we can perform two additions, one assuming the carryin is 1 and one assuming the carryin is 0, and then choose between the two results once the actual carryin is known. This scheme, proposed by Sklanski in 1960, is called conditional-sum addition [10]. An implementation of this scheme was first realized by Bedrij and is called the *Carry Select Adder* (CSLA) [11].

The CSLA divides the adder into blocks that have the same input operands except for the carryin. Figure 2.14 shows a possible implementation for a a 16-bit CSLA using ripple carry adder blocks. The carryout of the first block is used as the select line for the 9-bit 2-to-1 mux. The second and third blocks calculate the signals  $sum_{16} - sum_8$  in parallel, with one block having its carryin hardwired to 0 and another hardwired to 1. After one 8-bit ripple adder delay there is only the delay of the mux to choose between the results of block 2 or 3. Equation 2.26 shows the delay for this adder. The 16-bit CSLA can also be built by dividing it into even more blocks. Figure 2.15 shows the block diagram for the adder if it were divided into 4-bit RCA blocks. Equation 2.27 expresses the delay for this structure.

$$t_{CSLA_{16a}} = t_{8bitRCA} + t_{(9bitmux)} \quad (2.26)$$

$$t_{CSLA_{16b}} = t_{4bitRCA} + 3 \cdot t_{(5bitmux)} \quad (2.27)$$

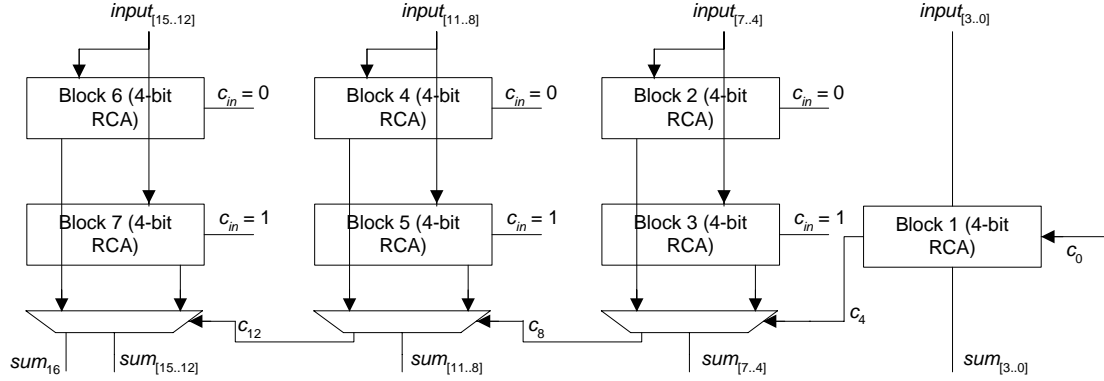


Figure 2.15: Schematic for a 16-bit CLSA with 4-bit RCA blocks

The CSLA can use any of the adder structures discussed in the previous sections as subcells. The delay ultimately comes down to the speed of the adder subcell used and the speed of the muxes used to select the sum bits. A general equation for this adder is expressed in Equation 2.28, where  $N$  is the adder size, and  $k$  is the group size of each adder subcell.

$$t_{CSLA_N} = t_{k\text{-bit adder}} + \frac{N}{k} \cdot t_{((k+1)\text{bit mux})} \quad (2.28)$$

The CSLA described so far is called the *Linear Carry Select Adder*, because its delay is linearly dependent on the length of the adder. In the worst case, the carry signal must ripple through each mux in the adder. Also, notice that the subcells are done with their addition at the same time, yet the more significant bits are waiting at the input of the mux to be selected. An optimization to this structure is to vary the length of each of the adder subcells, observing the fact that the later groups have more time to add because the select signal for their muxes take longer to arrive. The result is a structure called the *Square Root Carry Select Adder*, and Equation 2.29 expresses the delay equation [4], where  $t_{adder}$  is the delay of the first block which generates the select line for a mux, and  $\sqrt{2N}$  is the number of groups the CSLA is divided into. The derivation for this square root CSLA is done by Rabaey [4].

$$t_{sqCSLA_N} = t_{adder} + (\sqrt{2N}) \cdot t_{mux} \quad (2.29)$$

From a VLSI design perspective, the CSLA uses a large amount of area compared to the other adders. There is hardware in this architecture which computes results that are thrown away on every addition, but the fact that the delay for an add can be replaced by the delay of a mux makes this architecture very fast. Also, the Linear CSLA has regularity that makes it easier to layout.

| Structure | FA cell         |            | Bypass logic |            | 8xFAs | 2xBypass | Total Trans. |
|-----------|-----------------|------------|--------------|------------|-------|----------|--------------|
|           | <i>carryout</i> | <i>sum</i> | 4-input AND  | 2-to-1 mux |       |          |              |
| RCA       | 12              | 16         | –            | –          | 224   | –        | 224          |
| CSKA      | 12              | 16         | 10           | 6          | 224   | 32       | 256          |

Table 2.4: Transistor count for 8-bit RCA and CSKA adders

| Structure | PFA cell   |                      | CLA logic<br>1 <sup>st</sup> level |                       |                       |                       |                    |                    | CLA logic<br>2 <sup>nd</sup> level |                       |                       | 5-bit<br>2-to-1 mux |
|-----------|------------|----------------------|------------------------------------|-----------------------|-----------------------|-----------------------|--------------------|--------------------|------------------------------------|-----------------------|-----------------------|---------------------|
|           | <i>gen</i> | <i>prop/<br/>sum</i> | <i>c</i> <sub>1</sub>              | <i>c</i> <sub>2</sub> | <i>c</i> <sub>3</sub> | <i>c</i> <sub>4</sub> | <i>group<br/>g</i> | <i>group<br/>p</i> | <i>c</i> <sub>1</sub>              | <i>c</i> <sub>2</sub> | <i>c</i> <sub>3</sub> |                     |
| CLA       | 6          | 16                   | 10                                 | 18                    | 28                    | –                     | 28                 | 10                 | 10                                 | 18                    | 28                    | –                   |
| CSLA      | 6          | 16                   | 10                                 | 18                    | 28                    | 40                    | –                  | –                  | –                                  | –                     | –                     | 22                  |

| Structure     | 4 PFAs | 4-bit CLA adder | No. of 4-bit adders | Total Tx's |
|---------------|--------|-----------------|---------------------|------------|
| CLA (cont'd)  | 88     | 182             | 2                   | 420        |
| CSLA (cont'd) | 88     | 184             | 3                   | 574        |

Table 2.5: Transistor Count for 8-bit CLA adder and CSLA adders

The Square Root CSLA, on the other hand, has higher performance but is more time consuming to implement. The varying length of the adders makes subcell reuse difficult. Rabaey [4] demonstrates a Square Root CSLA with subsequent adder blocks increasing by one bit. In practice, using a high performance subcell such as the CLA adder in the Square Root CSLA will result in subsequent blocks which differ by more than one bit. For example, in a 12-bit Square Root CSLA, the first block will consist of a 4-bit CLA adder, and the second and third blocks will consist of two 8-bit CLA adders, followed by a mux. This may not provide as much speed up as an optimized Square Root CSLA, but it requires less time to implement.

## 2.3 Algorithm Analysis

In this section, we examine the algorithms of the previous sections and choose an architecture that meets the needs of AsAP [5]. Each processor in AsAP needs a fast, 16-bit signed adder for its arithmetic logic unit. There are other structures in each processor that require an adder, such as the address generators, program counter, repeat instruction hardware, and the multiply-accumulate unit. Hence, another criterion besides high performance is choosing an adder architecture that is modular. The ability to change the length of the adder without too much modification or time allows reuse and reduces design time.

Tables 2.4 and 2.5 show the transistor counts for implementations of an 8-bit adder amongst

| Adder | <i>Delay</i>             | <i>Trans. Count</i> | <i>Area</i> | <i>Design Time</i> |
|-------|--------------------------|---------------------|-------------|--------------------|
| RCA   | N                        | 224                 | 1           | 1                  |
| CSKA  | $N/k$                    | 256                 | 1.14        | 2                  |
| CLA   | $\log_4 N$               | 420                 | 1.88        | 8                  |
| CSLA  | $\frac{\log_4 N}{(N/k)}$ | 574                 | 2.56        | 10                 |

Table 2.6: Adder Comparison:  $N$  is the length of the adder and  $k$  is the group size, if applicable

the different adder architectures. For the CSKA we assume a group size of 4 using an RCA as a subcell. For the CSLA, we assume the use of three 4-bit CLA adders as subcells, with the carryout of the first CLA adder used as the select line for a pass gate mux. Table 2.6 compares the different algorithms. The *Delay* column expresses how the delay of the adder is proportional to the length. For the CSLA, this assumes that its subcells use the CLA adder. The next column, *Trans. Count*, lists the number of transistors for an 8-bit adder, and are taken from the values of Table 2.4 and 2.5. The purpose of this column is to obtain a general idea of the sizes of these adders. This column considers only the number of transistors, and does not take into account the sizes of the transistors or the extra space required to wire the unit. The next column, *Area*, normalizes the area for the RCA (based on the transistor count) and compares the relative sizes of the other adders to this normalized value. And finally, the *Design Time* column is an estimate of the time required to design and layout the particular adder based on layout we have done. It normalizes the time based on the RCA design and layout time.

The results from Table 2.6 illustrate the area/speed tradeoff in digital circuits. Since ASAP requires high performance, we are willing to trade off some area for speed. The CLA adder has delay that grows logarithmically, and it has the regularity that will allow us to adjust the size of the adder without much additional design time. It is for these reasons that we choose to use the CLA architecture.

## 2.4 Implementation of a 16-bit Carry Look Ahead Adder

This section describes the design, implementation, and results of building a 16-bit signed CLA adder. The ASAP will be built in silicon in the TSMC 0.18  $\mu\text{m}$  process. The technology files for the tools we use follow the design rules for that technology, and the results presented are in 0.18  $\mu\text{m}$ . We follow the following design flow:

1) Program the CLA adder in a high level Hardware Description Language (HDL). We write the

|                | Signals from 4 PFAs |       |       |       |       |       |       |       |
|----------------|---------------------|-------|-------|-------|-------|-------|-------|-------|
|                | $p_0$               | $p_1$ | $p_2$ | $p_3$ | $g_0$ | $g_1$ | $g_2$ | $g_3$ |
| Signals driven | 4                   | 6     | 6     | 4     | 4     | 3     | 2     | 1     |

Table 2.7: Number of signals driven by each *propagate* and *generate* signal in the 4-bit CLA logic

code in Verilog and test it using NC-Verilog [12].

- 2) Design the adder at the transistor level using HSPICE [13] to obtain transistor sizing. This step also introduces a first estimate of the speed of the adder.
- 3) Layout the adder using MAGIC [14]. This step determines the area.
- 4) Functionally simulate the adder using IRSIM [15].
- 5) Extract the layout to a spice netlist file, and test the performance in HSPICE.

After the 5<sup>th</sup> step, it is often necessary to resize some transistors in the critical path to achieve higher performance. This step is repeated until the additional time required to improve the adder results in minimal gains. Results for speed and area and their operating parameters are presented in Section 2.4.6.

Loads are placed on the outputs of the adder to simulate real circuit conditions. The load placed on each output is usually 15 fF, or the equivalent input capacitance of four minimum-sized inverters.

### 2.4.1 Partial Full Adders

The logic for a PFA is presented in Section 2.1.3. It consists of only one AND gate and two XOR gates logically, but it is often implemented using one NAND, one XOR gate, one XNOR, and three inverters. Figure 2.17 shows the logic gates necessary to build the PFA. The two inverters for the *propagate* signal and *sum* signal give additional driving ability. The XOR and XNOR gates are each implemented two ways with transmission gates (as shown in Figure 2.16). The inverter following the NAND gate is there to implement the AND function, but also serves the benefit of increasing driving ability for the *generate* signal. Table 2.7 shows the number of signals each *propagate* and *generate* need to drive in the 4-bit CLA logic. This table illustrates that the loads on the outputs of a PFA are significant, hence the need for additional inverters to drive these loads.

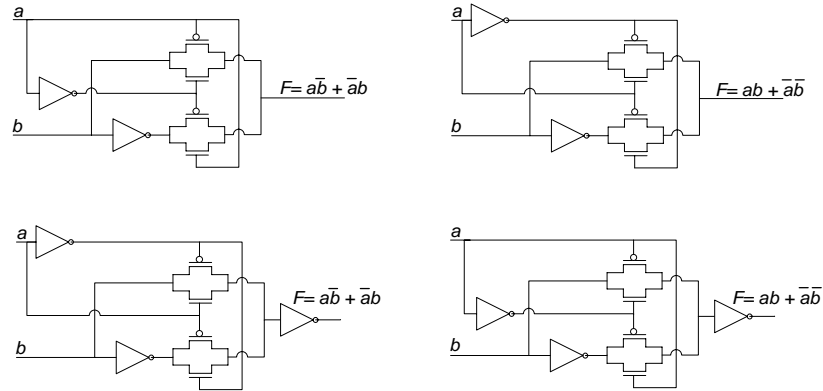


Figure 2.16: Schematic for XOR and XNOR gates: The two figures on the left implement the XOR function. The two figures on the right implement the XNOR function. The two figures in the top row implement their respective functions without an inverter driving the output, thus taking only 8 transistors to build. The two figures in the bottom row use an inverter at their outputs and have additional driving power, thus requiring 10 transistors to build. The XOR with a driver uses the XNOR without a driver, and vice versa.

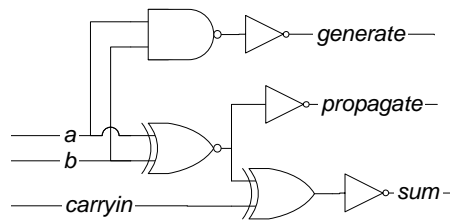


Figure 2.17: Gate Schematic for the implemented PFA

## 2.4.2 Carry look ahead logic for 4 bits

The CLA logic for the 4-bit CLA adder calculates the signals  $c_1$ ,  $c_2$ ,  $c_3$ ,  $Gto16$  (*group generate*), and  $Pto16$  (*group propagate*). The gate schematic is presented in Figure 2.10 and is implemented as shown in that figure. Its inputs arrive from the *propagate* and *generate* signals from four PFAs, and it drives carries into three PFAs and its *group propagate* and *generate* signals into the next level of CLA logic.

## 2.4.3 4-bit CLA adder

The 4-bit CLA adder is used as the adder subcell for the 16-bit CLA adder. This subcell connects 4 PFAs to one level of CLA logic, and since it will be used to build a larger adder, it does not calculate the *carryout* (5<sup>th</sup> sum bit). Since there is no advantage to having one sum bit arrive earlier than another, the transistors are sized to equate the sum delays as much as possible. Each

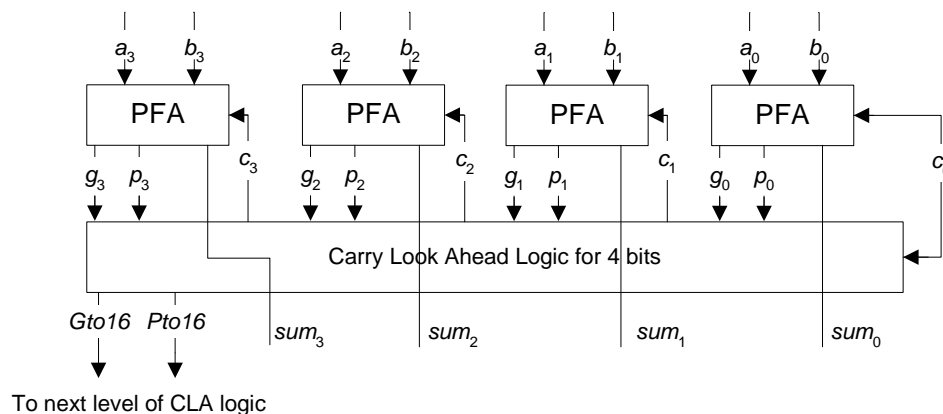


Figure 2.18: Block Diagram for the implemented 4-bit CLA adder

PFA is identical, so we size the gates that calculate the  $c_1$ ,  $c_2$ , and  $c_3$  signals to have equal delay. Figure 2.18 shows the block diagram for the implemented 4-bit CLA adder.

#### 2.4.4 16-bit CLA adder

The 16-bit 2's complement CLA adder consists of four 4-bit CLA adders and a second level of CLA logic to generate the group carries for these subcells. A group size of four is chosen because of the limits on the speed of high fan-in gates. The maximum fan-in is five, and the  $c_4$  signal contains two gates with this fan-in. Also, a group size of four is of small enough granularity to allow us to build any size adder that is a multiple of four. If an application calls for an adder that is not a multiple of four, we simply tie the lower or upper bit(s) to ground and ignore their respective sum bits.

The schematic for this adder is also shown in Figure 2.12. The figure is for an unsigned adder, and the implementation for the AsAP processor requires a signed adder. Only a slight modification needs to be made to the unsigned adder to make it a signed adder. Two 2-input XOR gates are appended to the most significant portion of the adder to implement sign extension. Figure 2.19 shows a schematic of the two XOR gates with their inputs. These gates implement the logic to calculate the sum of the most significant bit, identical to the sum hardware for a full adder in Section 2.1.2. Figure 2.20 shows our implementation of the 16-bit signed adder.

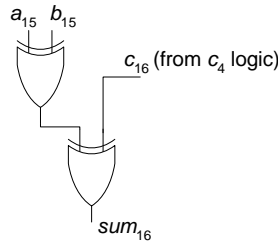


Figure 2.19: Two XOR gates for sign extension, appended to the unsigned adder to make a signed adder.

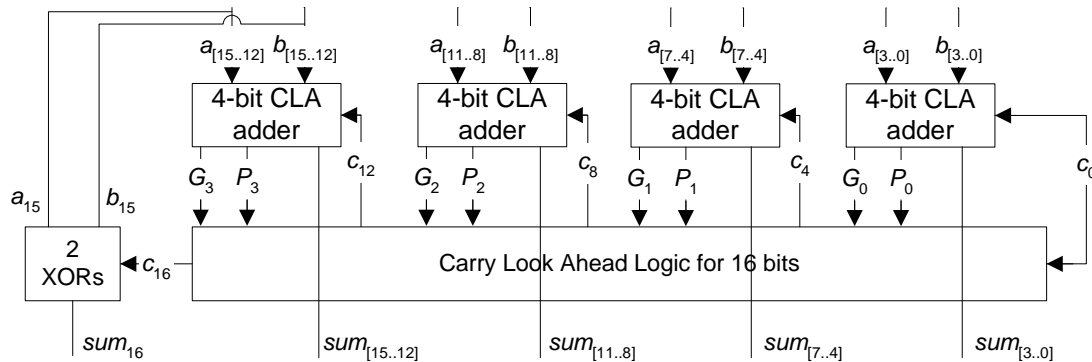


Figure 2.20: A 16-bit signed CLA adder

### 2.4.5 Critical Path Analysis

There is a discussion of the critical path for the 16-bit CLA adder near the end of Section 2.2.3. To exercise the critical path of the signed adder, we choose a set of input vectors that will cause a transition on the *group generate* signal of one of the 4-bit CLA adders, and a transition on the  $c_{12}$  signal in the second level of CLA logic. These two signals have two 4-input NAND gates in their logic and we want the change in inputs to cause transitions through these gates. Table 2.8 shows a list of the input signals and the transitions they make to exercise the critical path. Figure 2.21 reflects the values that are calculated from these inputs. It shows the *group generate* and *propagate* logic for the four adder subcells and their outputs feeding into the logic for  $c_3$ . This  $c_3$  signal is the carryin ( $c_{12}$ ) to the fourth adder subcell, and will be the critical signal because of its two 4-input NAND gates.

Although the  $c_4$  logic, used to calculate  $c_{16}$ , contains two 5-input NAND gates, it only drives an XOR gate. The other carryout signals drive the carryin signals for the 4-bit CLA adders, which present a much greater load than just one XOR gate. This is the reason we target the  $c_3$  logic

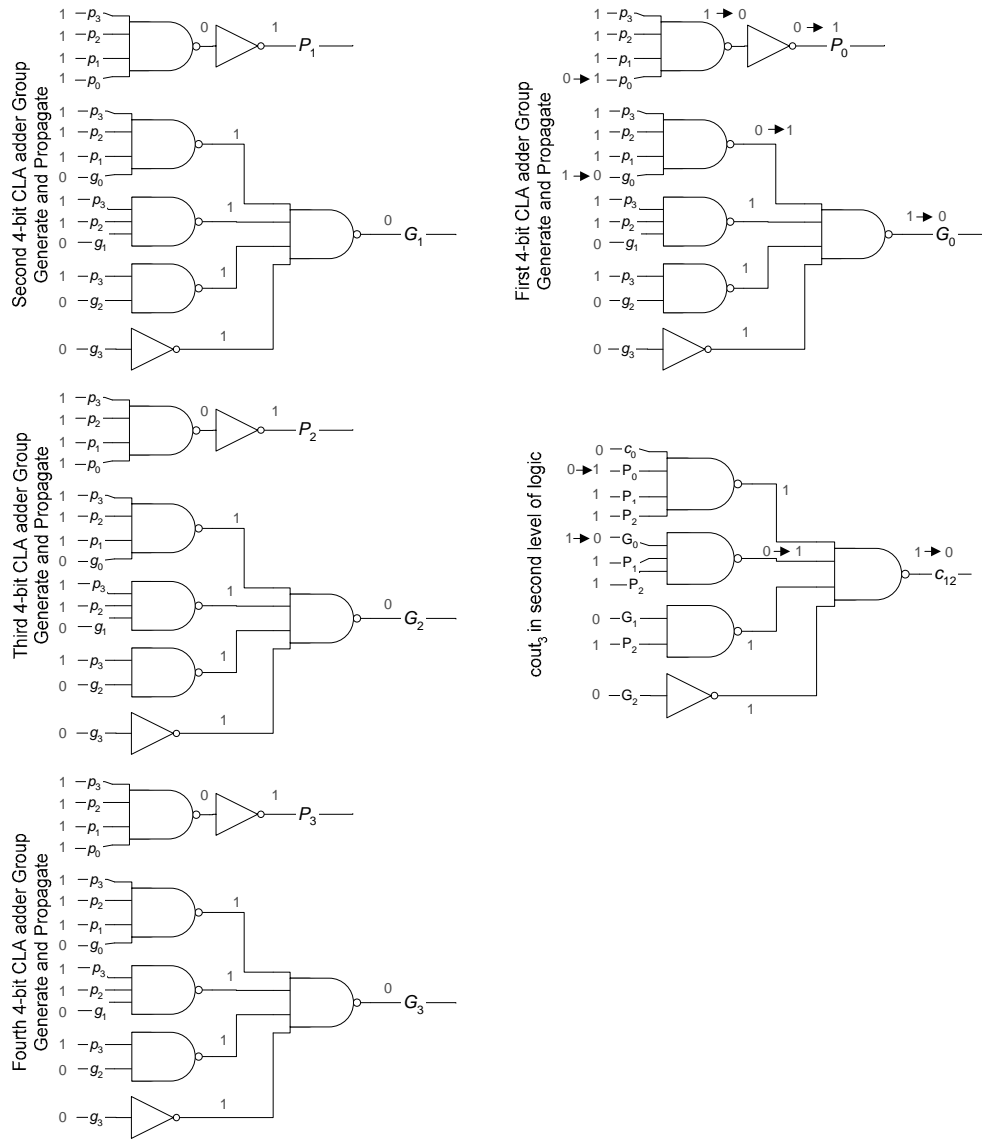


Figure 2.21: Schematic of transitions for the 16-bit CLA adder: This schematic shows the signals in the group propagate and generate logic (which are named  $G$  and  $P$ ) for each of the four 4-bit CLA adders. The  $cout_3$  signal in the second level of logic uses these group propagate and generate signals.

| Inputs                  |              | Local generate and propagate signals           | Group generate and propagate signals | Carryout signal                              |
|-------------------------|--------------|--|--------------------------------------|--|
| $a_{15} = 0$            | $b_{15} = 1$ | $p_3 = 1, g_3 = 0$                             | $P_3 = 1$                            | $c_{12} = 1 \rightarrow 0$<br>( $c_3$ logic) |
| $a_{14} = 0$            | $b_{14} = 1$ | $p_3 = 1, g_3 = 0$                             | $G_3 = 0$                            |  |
| $a_{13} = 0$            | $b_{13} = 1$ | $p_3 = 1, g_3 = 0$                             |                                      |  |
| $a_{12} = 0$            | $b_{12} = 1$ | $p_3 = 1, g_3 = 0$                             |                                      |  |
| $a_{11} = 0$            | $b_{11} = 1$ | $p_3 = 1, g_3 = 0$                             | $P_2 = 1$                            |  |
| $a_{10} = 0$            | $b_{10} = 1$ | $p_3 = 1, g_3 = 0$                             | $G_2 = 0$                            |  |
| $a_9 = 0$               | $b_9 = 1$    | $p_3 = 1, g_3 = 0$                             |                                      |  |
| $a_8 = 0$               | $b_8 = 1$    | $p_3 = 1, g_3 = 0$                             |                                      |  |
| $a_7 = 0$               | $b_7 = 1$    | $p_3 = 1, g_3 = 0$                             | $P_1 = 1$                            |  |
| $a_6 = 0$               | $b_6 = 1$    | $p_3 = 1, g_3 = 0$                             | $G_1 = 0$                            |  |
| $a_5 = 0$               | $b_5 = 1$    | $p_3 = 1, g_3 = 0$                             |                                      |  |
| $a_4 = 0$               | $b_4 = 1$    | $p_3 = 1, g_3 = 0$                             |                                      |  |
| $a_3 = 0$               | $b_3 = 1$    | $p_3 = 1, g_3 = 0$                             | $P_0 = 0 \rightarrow 1$              |  |
| $a_2 = 0$               | $b_2 = 1$    | $p_3 = 1, g_3 = 0$                             | $G_0 = 1 \rightarrow 0$              |  |
| $a_1 = 0$               | $b_1 = 1$    | $p_3 = 1, g_3 = 0$                             |                                      |  |
| $a_0 = 1 \rightarrow 0$ | $b_0 = 1$    | $p_3 = 0 \rightarrow 1, g_3 = 1 \rightarrow 0$ |                                      |  |

Table 2.8: Input Signals and Transitions for the 16-bit CLA adder: The group propagate and generate signals are  $P$  and  $G$ , respectively, to distinguish them from the local propagate and generate signals,  $p$  and  $g$ .

in our critical path analysis.

## 2.4.6 Layout and Performance

The main cells that the 16-bit CLA signed adder uses are the 4-bit CLA adders and the CLA logic. The 4-bit CLA adders, in turn, use four partial full adders and one level of CLA logic. The layout of the 4-bit CLA subcell occupies  $1011 \mu\text{m}^2$  with dimensions of  $55 \mu\text{m}$  by  $19 \mu\text{m}$  in TSMC  $0.18 \mu\text{m}$  technology. The 16-bit CLA signed adder layout occupies  $6734 \mu\text{m}^2$  with dimensions of  $219.5 \mu\text{m}$  by  $31 \mu\text{m}$ . A total of four metal layers in a six metal process are used. We reserve the upper two layers for signals such as the global power and ground, the clock, and other global signals that may need to be routed over the cell.

After laying out the adder, it is extracted from MAGIC into an IRSIM netlist and tested for functionality. There are 908 transistors in the 16-bit signed adder (as reported by IRSIM). The layout is then extracted into an HSPICE netlist. An HSPICE driver file is written and the input pulses are generated. These pulses go through buffers (two inverters in series) before being fed to the inputs of the adder. The inverters for the buffers are sized with a PMOS to NMOS ratio of 2.6 to 1, with the actual size of the PMOS =  $1.17 \mu\text{m}$  and NMOS =  $0.45 \mu\text{m}$  (this sizing results in an equal rise/fall time). The adder is tested with the input  $a = 0000000000000001$ ,  $b = 1111111111111111$ , and  $c_0 = 0$ , while toggling the least significant bit of  $a$ . The 17 sum bits are each loaded with

|                    | 1.0 Volts |      |      | 1.8 Volts |      |      |
|--------------------|-----------|------|------|-----------|------|------|
|                    | 25°C      | 40°C | 55°C | 25°C      | 40°C | 55°C |
| $t_{prop}$ rise    | 2413      | 2427 | 2446 | 831       | 854  | 869  |
| $t_{prop}$ fall    | 2252      | 2268 | 2285 | 788       | 815  | 833  |
| $t_{prop}$ average | 2333      | 2348 | 2366 | 810       | 835  | 851  |

Table 2.9: 16-bit CLA adder results, in picoseconds: The times in this table reflect the worst case propagation delay of the sum. The conditions for these tests are the use of Typical NMOS and Typical PMOS transistors from the spice technology file from the MOSIS TSMC 0.18  $\mu\text{m}$  process. The typical operating voltage is 1.8 V, but operation at lower voltages is also expected, so we test the circuit at 1.0 V to check robustness and speed at low voltage. We found that the worst case propagation delay occurred with the input  $a = 1111111111111111$ ,  $b = 0000000000000001$ , and  $c_0 = 0$ , while toggling the least significant bit of  $b$ . The extraction threshold is 0.1 fF.

a 15 fF capacitor. The latest sum bit to arrive is  $sum_{15}$ . Since the same critical path will be exercised if we swap the input vectors from  $a$  to  $b$  and  $b$  to  $a$ , we also test the adder with the input  $a = 1111111111111111$ ,  $b = 0000000000000001$ , and  $c_0 = 0$ , while toggling the least significant bit of  $b$ . We attempted to improve the speed by implementing the 4-input NAND gate for the  $c_3$  logic using two levels of logic, but this did not show any improvement and actually slowed down the addition operation. Table 2.9 summarizes the performance of the adder under different conditions. When determining the worst case delay, we looked at the waveforms for the 16-bit sum and measured the delay of the latest signal (usually either  $sum_{14}$  or  $sum_{15}$ ).

From the HSPICE netlist, we also run simulations to calculate the energy consumed per addition. The tests are run with  $a = 1111111111111111$ ,  $b = 0000000000000001$ , and  $c_0 = 0$ , while toggling the least significant bit of  $b$ . We take the average current measurement over two additions (once for all the outputs transitioning high, and once for all the outputs transitioning low), allowing ample time for each addition operation to take place. The propagation delays will be different for the two different additions, so we average those as well. We run the adder at different supply voltages and report the energy consumed and time it takes for the addition operation to complete. This will provide us with an important metric, the *energy-delay product*, which provides a measure of both performance and energy [4]. Table 2.10 is a table of the energy-delay product for addition operations at different supply voltages. It shows the optimum energy-delay product at a supply voltage of  $\sim 1.2$  V. The plot of normalized delay, energy, and energy-delay versus supply voltage is in Figure 2.22.

Figure 2.23 shows the layout for the 16-bit signed adder.

| Supply Voltage (V) | Time Interval ( $t$ ) | Avg. Current ( $i$ ) | Charge ( $Q = t \times i$ ) | Energy Consumed ( $E = Q \times V$ ) | Avg. Delay ( $d$ ) | Energy-delay Product ( $E \times d$ )        |
|--------------------|-----------------------|----------------------|-----------------------------|--------------------------------------|--------------------|--|
| 0.8                | 15 ns                 | 0.0668 mA            | 1.00 pC                     | 0.802 pJ                             | 4.56 ns            | $3.66 \times 10^{-21}$ J·s                   |
| 1.0                | 12 ns                 | 0.107 mA             | 1.28 pC                     | 1.28 pJ                              | 2.35 ns            | $3.02 \times 10^{-21}$ J·s                   |
| 1.2                | 12 ns                 | 0.133 mA             | 1.60 pC                     | 1.92 pJ                              | 1.52 ns            | <b><math>2.91 \times 10^{-21}</math></b> J·s |
| 1.4                | 12 ns                 | 0.157 mA             | 1.88 pC                     | 2.64 pJ                              | 1.16 ns            | $3.06 \times 10^{-21}$ J·s                   |
| 1.6                | 12 ns                 | 0.183 mA             | 2.20 pC                     | 3.51 pJ                              | 971 ps             | $3.41 \times 10^{-21}$ J·s                   |
| 1.8                | 12 ns                 | 0.212 mA             | 2.54 pC                     | 4.58 pJ                              | 835 ps             | $3.82 \times 10^{-21}$ J·s                   |

Table 2.10: Energy-delay product for the 16-bit signed adder

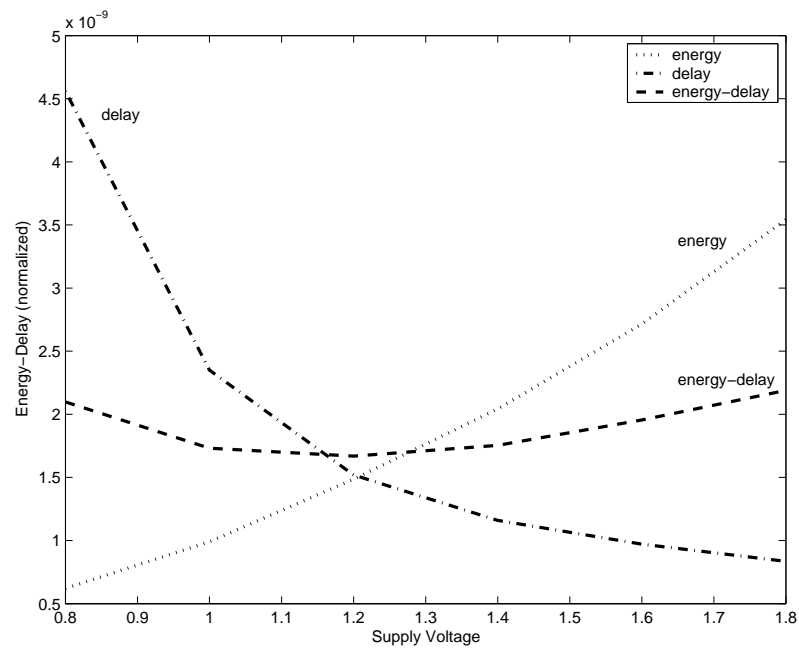
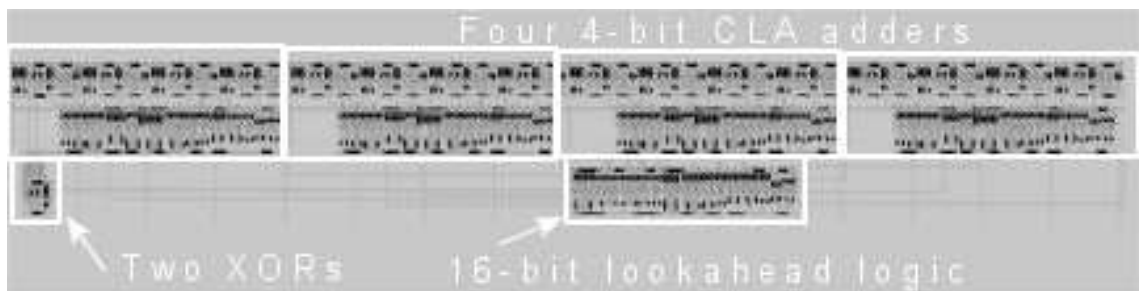
Figure 2.22: Energy-delay plot for the 16-bit signed adder. The optimum energy-delay product is at a supply voltage of  $\sim 1.2$  V. The plot is generated by Matlab [16].

Figure 2.23: Final layout for the 16-bit signed adder

## 2.5 Summary

This chapter presents various adder algorithms and analyzes them with respect to speed and area. A 16-bit CLA adder architecture is then chosen to be used in AsAP, and is measured at a speed of 854 ps at 40°C and 1.8 V, while consuming an area of 6734  $\mu\text{m}^2$ . It consumes 4.58 pJ per addition at 1.8 V. In general, larger architectures (larger area) consume more energy, with the tradeoff of higher performance. The CLA adder probably consumes more energy than the RCA and CSKA because of its higher transistor count. However, this architecture provides good performance and modularity for adders of different lengths, so it is the reason we choose it for AsAP.

## Chapter 3

# Multiplication Schemes

Multiplication hardware often consumes much time and area compared to other arithmetic operations. Digital signal processors use a multiplier/MAC unit as a basic building block [3] and the algorithms they run are often multiply-intensive. In this chapter, we discuss different architectures for multiplication and the methods that improve speed and/or area. Also, it is important to consider these methods in the context of VLSI design. It is beneficial to find structures that are modular and easy to layout. Many of the architectures described in this chapter will be used in the implementation of the multiply-accumulate unit for AsAP [5].

### 3.1 Multiplication Definition

To perform an  $M$ -bit by  $N$ -bit multiplication as shown in Figure 3.1, the  $M$ -bit multiplicand  $A = a_{(M-1)}a_{(M-2)}\dots a_1a_0$  is multiplied by the  $N$ -bit multiplier  $B = b_{(N-1)}b_{(N-2)}\dots b_1b_0$  to produce the  $M+N$ -bit product  $P = p_{(M+N-1)}p_{(M+N-2)}\dots p_1p_0$ . The unsigned binary numbers,  $A$  and  $B$ , can be expressed by Equations 3.1 and 3.2. The equation for the product is defined in Equation 3.3 [17].

$$A = \sum_{i=0}^{M-1} a_i \cdot 2^i \quad (3.1)$$

$$B = \sum_{i=0}^{N-1} b_i \cdot 2^i \quad (3.2)$$

$$P = A \cdot B = \left( \sum_{i=0}^{M-1} a_i \cdot 2^i \right) \left( \sum_{j=0}^{N-1} b_j \cdot 2^j \right) = \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} (a_i \cdot b_j \cdot 2^{i+j}) \quad (3.3)$$

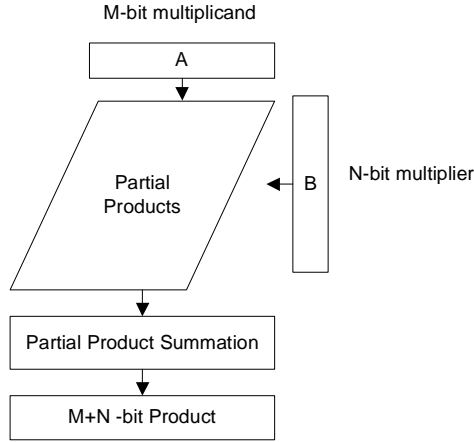


Figure 3.1: Generic Multiplier Block Diagram

With two's complement multiplication, both numbers are signed and the result is signed. If  $A$  and  $B$  are signed binary numbers, they are expressed by Equations 3.4 and 3.5. The equation for the product is defined in Equation 3.6.

$$A = -a_{M-1} \cdot 2^{M-1} + \sum_{i=0}^{M-2} a_i \cdot 2^i \quad (3.4)$$

$$B = -b_{N-1} \cdot 2^{N-1} + \sum_{i=0}^{N-2} b_i \cdot 2^i \quad (3.5)$$

$$P = A \cdot B = (-a_{M-1} \cdot 2^{M-1} + \sum_{i=0}^{M-2} a_i \cdot 2^i)(-b_{N-1} \cdot 2^{N-1} + \sum_{j=0}^{N-2} b_j \cdot 2^j) \quad (3.6)$$

A multiplication operation can be broken down into two steps:

- 1) Generate the partial products.
- 2) Accumulate (add) the partial products.

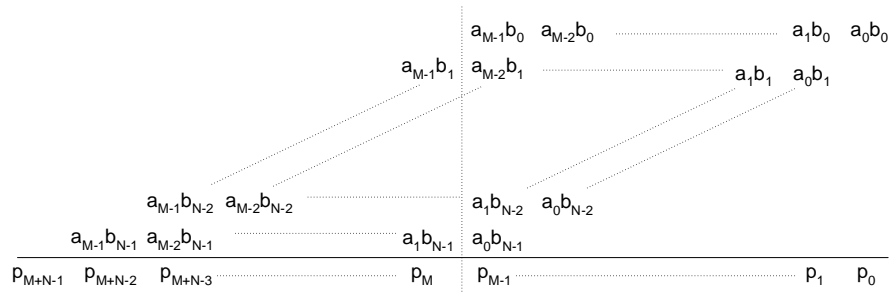


Figure 3.2: Partial product array for an  $M \times N$  multiplier: As shown it implies that  $M = N$

### 3.2 Array Multiplier

From Equation 3.3, each multiplicand is multiplied by a bit in the multiplier, generating  $N$  partial products. Each of these partial products is either the multiplicand shifted by some amount, or 0. This is illustrated in Figure 3.2 for an  $M \times N$  multiply operation. This figure can be mapped directly into hardware and is called the array multiplier. The generation of partial products consists of simple AND'ing of the multiplier and the multiplicand. The accumulation of these partial products can be done with rows of ripple adders. Thus, the carry out from the least significant bit ripples to the most significant bit of the same row, and then down the “left side” of the structure.

Figure 3.3 shows the generation of the partial product bits for a  $4 \times 4$ -bit multiplier. Figure 3.4 shows a  $4 \times 4$  unsigned array multiplier [4]. The partial products are added in ripple fashion with half and full adders. A full adder's inputs require the carryin from the adjacent full adder in its row and the sum from a full adder in the above row. Rabaey [4] states that finding the critical path in this structure is non-trivial, but once identified, results in multiple critical paths. It requires a lot of time to optimize the adders in the array since all adders in the multiple critical paths need to be optimized to result in any speed increase (this implies optimization of both the sum and carryout signals in a full adder). The delay basically comes down to a ripple delay through a row, and then down a column, so it is linearly proportional ( $t_d \approx (M + N)$ ) to the sum of the sizes of the input operands.

Array multipliers can also be built with “carry-save” adders (CSAs) to avoid the “rippling” effect [18]. These adders bypass the addition of the carry until the very last stage. A full adder can be used as a counter, in the sense that its output is the number of 1's at the input. It reduces the number of bits from 3 to 2. A half adder simply moves bits, from two with a weight  $2^i$  to one with a weight  $2^i$  and one with a weight  $2^{i+1}$ . Fig 3.5 shows the dot diagrams for the half adder and full

|       |       |       |          |          |          |          |       |
|-------|-------|-------|----------|----------|----------|----------|-------|
|       |       |       | $a_3b_0$ | $a_2b_0$ | $a_1b_0$ | $a_0b_0$ |       |
|       |       |       | $a_3b_1$ | $a_2b_1$ | $a_1b_1$ | $a_0b_1$ |       |
|       |       |       | $a_3b_2$ | $a_2b_2$ | $a_1b_2$ | $a_0b_2$ |       |
|       |       |       | $a_3b_3$ | $a_2b_3$ | $a_1b_3$ | $a_0b_3$ |       |
| $p_7$ | $p_6$ | $p_5$ | $p_4$    | $p_3$    | $p_2$    | $p_1$    | $p_0$ |

Figure 3.3: Partial product array for a 4x4 multiplier

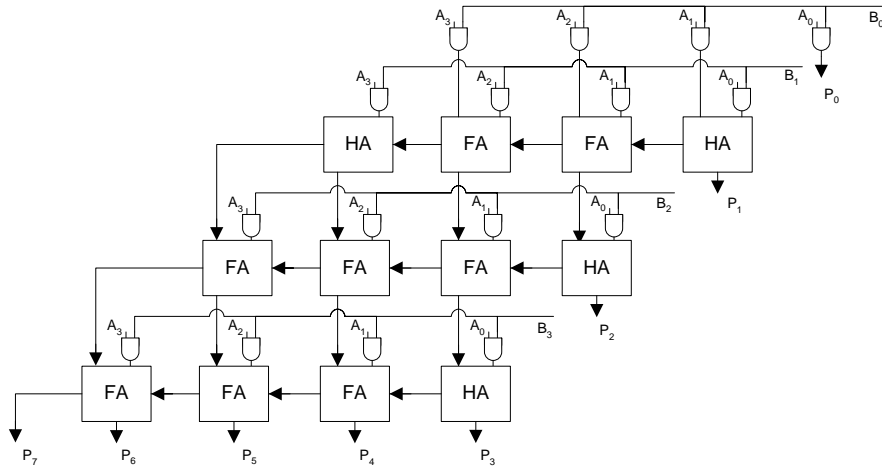


Figure 3.4: Array multiplier block diagram for a 4x4 multiplier [4]

adder. Each dot represents a bit, and both these adders can be used as carry-save adders. For an array multiplier, each row of CSAs (full adders) adds an additional partial product, and since there is no carry propagation, the delay is only dependent on the length of the multiplier ( $t_d \approx N$ ).

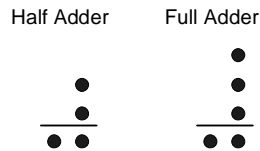


Figure 3.5: Dot Diagrams for a half adder and a full adder

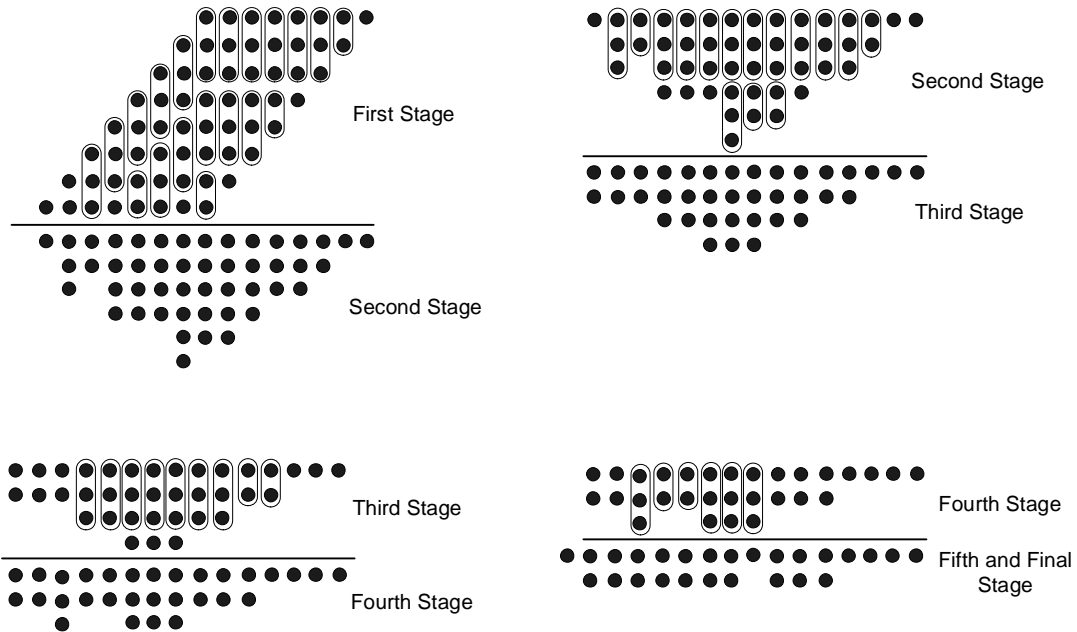


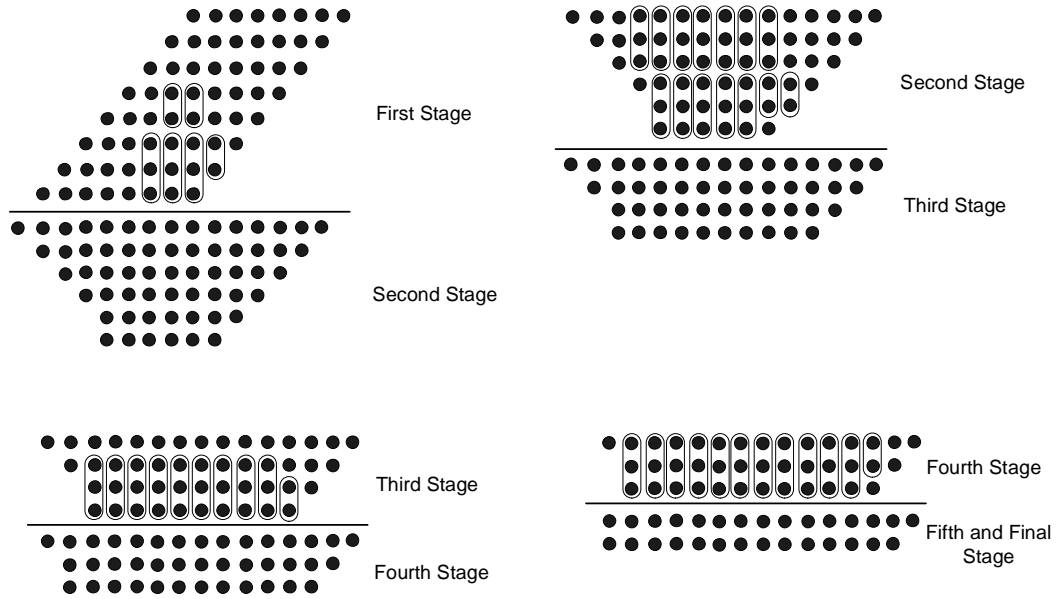
Figure 3.6: Wallace Tree for an  $8 \times 8$ -bit partial product tree

### 3.3 Tree Multiplier

The tree multiplier reduces the time for the accumulation of partial products by adding all of them in parallel, whereas the array multiplier adds each partial product in series. The tree multiplier commonly uses CSAs to accumulate the partial products.

#### 3.3.1 Wallace Tree

The reduction of partial products using full adders as carry-save adders (also called 3:2 counters) became generally known as the “Wallace Tree” [19]. This architecture reduces the partial products at a rate of  $\log_{\frac{3}{2}}(\frac{N}{2})$ . Figure 3.6 shows an example of tree reduction for an  $8 \times 8$ -bit partial product tree. The ovals around the dots represent either a full adder (for three circled dots) or a half adder (for two circled dots). This tree is reduced to two rows for a carry-propagate adder after four stages. There are many ways to reduce this tree with CSAs, and this example is just one of them.

Figure 3.7: Dadda Tree for an  $8 \times 8$ -bit partial product tree

| Tree type | Stages | FAs | HAs | registers |
|-----------|--------|-----|-----|-----------|
| Wallace   | 4      | 38  | 9   | 141       |
| Dadda     | 4      | 35  | 7   | 179       |

Table 3.1: Wallace and Dadda Comparison for an  $8 \times 8$ -bit partial product tree

### 3.3.2 Dadda Tree

Dadda introduced a different way to reduce the partial product tree which resulted in more efficient addition than the Wallace Tree [20]. Whereas Wallace uses CSAs to cover as many bits as possible in each stage, Dadda combines the bits as late as possible. Each stage in the tree is no more than 1.5 times the height of its successor, and CSAs are used as necessary to reduce the height of the next stage using that guideline. Figure 3.7 shows an  $8 \times 8$ -bit multiplier reduced using Dadda's method. The bits are reduced to carry-propagate form in 4 stages as with the Wallace tree. The Dadda tree delay also reduces at a rate of  $\log_{\frac{3}{2}}(\frac{N}{2})$ , but with less hardware. Table 3.1 compares the Dadda and Wallace implementations, showing the number of stages before the carry-propagate adder, the number of full adders used, the number of half adders used, and the number of registers used if the tree is pipelined (one pipe stage for each stage in the tree).

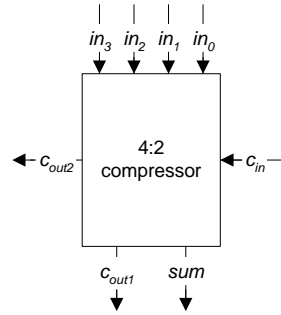


Figure 3.8: 4:2 compressor block diagram

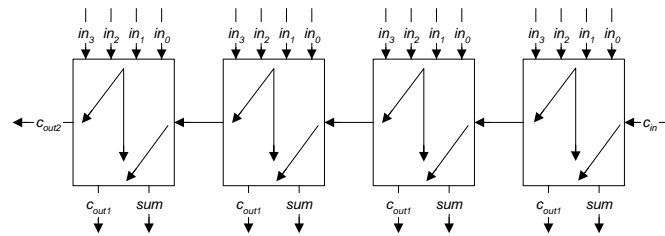


Figure 3.9: A chain of 4:2s

### 3.3.3 4:2 Carry Save Compressor

The disadvantage for both the Wallace and Dadda structures from a VLSI perspective is the irregularity in connections between the stages. Weinberger [21] introduced a structure he called “4:2 carry-save adder module” which takes in four inputs of weight  $2^i$ , one input of weight  $2^{i-1}$ , and produces one output of weight  $2^i$  and two outputs of weight  $2^{i+1}$ . Figure 3.8 shows the block diagram for a 4:2 compressor. This structure actually takes 5 inputs and produces 3 outputs, where one input is a carryin and one output is a carryout. These exist to allow the chaining together of 4:2s for rows of bits. The two carryouts generated are of the same weight, and the important feature is that there is no “rippling” effect from the carryin to the  $2^{nd}$  carryout. For a given set of inputs  $in_3, in_2, in_1$ , and  $in_0$ , the signal  $c_{out2}$  must not change when  $c_{in}$  changes. This is the reason it is an effective carry save structure, and it maintains the speed in reducing a tree of partial product bits.

Figure 3.9 shows a row of 4:2s, chained together, where the carryout of one 4:2 is fed into the carryin of another 4:2. The arrows inside each 4:2 illustrate how the outputs should depend on the inputs. Notice how the carryout to another 4:2 is not dependent on its carryin.

There are various ways to implement the 4:2 compressor, the simplest of which is cas-

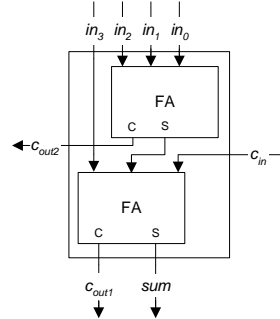


Figure 3.10: A 4:2 compressor made of 2 chained full adders

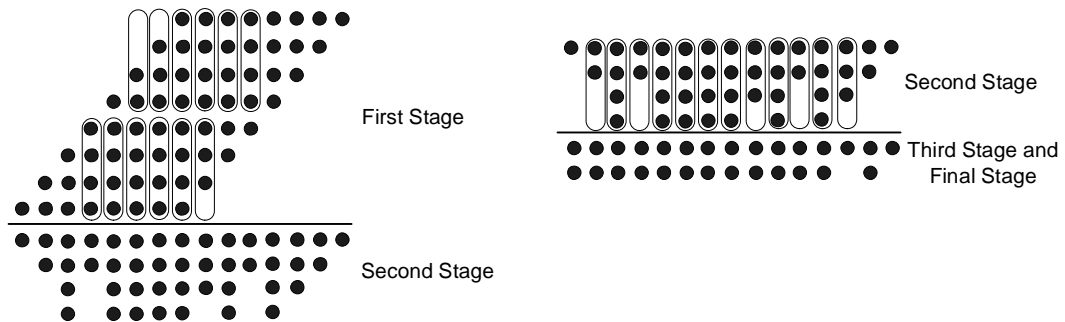


Figure 3.11: 4:2 compressor tree for an  $8 \times 8$ -bit multiplication

cading two full adder cells together. This structure was popularized by Santoro [18], who built a  $64 \times 64$ -bit multiplier. Figure 3.10 shows two full adders cascaded together to build a 4:2 cell. The interconnection of these full adders must not make the  $2^{nd}$  carryout dependent on the carryin.

We now show how rows of 4:2 compressors can be used to reduce the partial product bits in an  $8 \times 8$ -bit partial product tree. Figure 3.11 shows the rows of bits being reduced by 4:2s, and Table 3.2 now expands the previous comparison to include this result. Under the “FAs” column for the 4:2 entry, we multiply each 4:2 compressor in the tree by two, since it takes two full adders to

| Tree type | Stages | FAs | HAs | registers |
|-----------|--------|-----|-----|-----------|
| Wallace   | 4      | 38  | 9   | 141       |
| Dadda     | 4      | 35  | 7   | 179       |
| 4:2s      | 2      | 48  | 0   | 72        |

Table 3.2: Wallace, Dadda, and 4:2 Comparison for an  $8 \times 8$ -bit multiplier

build one 4:2. The speed of each 4:2 compressor is limited by the delay of 3 XOR gates in series [22]. Thus, a 4:2 is slightly faster than two full adders (which is four XOR gate delays). Thus, the tree compression is slightly faster using 4:2s with a tradeoff of a little more hardware. It reduces at a rate of  $\log_2(\frac{N}{2})$ , and there is also the added benefit of regular layout.

## 3.4 Partial Product Generation Methods

Another method to improve the speed of the multiplication operation is to improve the partial product generation step. This can be done in two ways:

- 1) Generate the partial products in a faster manner.
- 2) Reduce the number of partial products that need to be generated.

The first option can only be achieved if a different architecture for partial product generation is used. Considering that a bit is generated with just an AND gate delay (NAND followed by inverter in implementation), it seems that the partial product bits are already calculated in the fastest way possible. The second option will most likely take longer than an AND gate, but the reduced number of partial products results in a smaller tree and therefore reduces the time during the tree reduction step. As long as the time saved during tree reduction is greater than the extra time it takes to generate fewer partial products, it is beneficial to implement this option.

### 3.4.1 Booth's Algorithm

A simple method to reduce the number of partial products in a multiplication operation is called Booth's algorithm [23]. Through recoding the multiplier, potentially only half the number of partial products must be generated. The algorithm begins by looking at two bits of the multiplier at a time, and then determines what partial product to generate based on these two bits. This algorithm works for both signed and unsigned numbers. Table 3.3 shows Booth's algorithm and a decoding scheme. The decoder needs to select the correct partial product based on  $B_i$  and  $B_{i-1}$ . For a negative multiplicand, hardware that supports subtractions is needed, and for the positive multiplicand a 0 needs to be appended to the most significant bit to guarantee a positive result. Koren (Section 6.1, [9]) states two drawbacks to Booth's algorithm. One is that the variable number of add/subtract operations leads to a difficult design. For example, there are some cases where you must add all the partial products (all non-zero partial products generated), and other cases where only two non-zero partial products are generated, hence only one add operation is needed. The second drawback is the inefficiency of the algorithm when there are isolated 1's. For example, in

| Multiplier Bits<br>$B_i B_{i-1}$ | Explanation                  | Partial Product<br>Generated |
|----------------------------------|------------------------------|------------------------------|
| 00                               | string of 0's                | 0                            |
| 01                               | end of a string of 1's       | $A \cdot 2^i$                |
| 10                               | beginning of a string of 1's | $-A \cdot 2^i$               |
| 11                               | string of 1's                | 0                            |

Table 3.3: Original Booth Algorithm

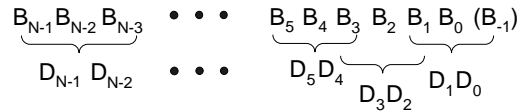


Figure 3.12: Recoded multiplier using Modified Booth Encoding

an  $8 \times 8$ -bit multiplication operation, a multiplier of 01010101 would normally generate 4 non-zero partial products (each 1 in the multiplier replicates the multiplicand), but with Booth's algorithm a total of 8 non-zero partial products are generated. In hardware, the design must always take into account the worst case, so with this algorithm there are still potentially 8 partial products that will be generated. This results in no improvement in terms of the hardware that needs to be built to add these 8 partial products (as compared to just the simple AND'ing of the multiplicand and multiplier).

### 3.4.2 Modified Booth's Algorithm

MacSorley [24] solved the drawbacks of Booth's algorithm by presenting an improved version which is popularly known as Modified Booth Encoding (also Radix-4 Modified Booth Encoding). It recodes two multiplier bits at a time, and uses a third bit as a reference. Figure 3.12 illustrates the manner in which the bits are recoded, with  $B = b_{(N-1)}b_{(N-2)}\dots b_1b_0$  as the original multiplier and  $D = d_{(N-1)}d_{(N-2)}\dots d_1d_0$  as the new recoded multiplier. Table 3.4 shows what the recoded multiplier values are, and the partial products they generate. The Booth decoder takes these recoded multiplier bits and selects the correct partial product. Bewick [25] shows an implementation for this scheme, and a block diagram of this implementation is shown in Figure 3.13.

| Multiplier bits |           |           | Recoded bits |           | Partial Product |
|-----------------|-----------|-----------|--------------|-----------|-----------------|
| $B_i$           | $B_{i-1}$ | $B_{i-2}$ | $D_i$        | $D_{i-1}$ |                 |
| 0               | 0         | 0         | 0            | 0         | +0              |
| 0               | 0         | 1         | 0            | 1         | +A              |
| 0               | 1         | 0         | 0            | 1         | +A              |
| 0               | 1         | 1         | 1            | 0         | +2A             |
| 1               | 0         | 0         | 1            | 0         | -2A             |
| 1               | 0         | 1         | 0            | 1         | -A              |
| 1               | 1         | 0         | 0            | 1         | -A              |
| 1               | 1         | 1         | 0            | 0         | -0              |

Table 3.4: Modified Booth Algorithm:  $B$  is the original multiplier,  $D$  is the recoded multiplier, and  $A$  is the multiplicand. Note that it is easy to generate the  $2A$  multiples of  $A$ , for it is simply a left shift of the multiplicand.  $B_i$  is the *sign* bit, and determines whether or not to invert the partial product bit. It is also added into the least significant position of the partial product that it generates.

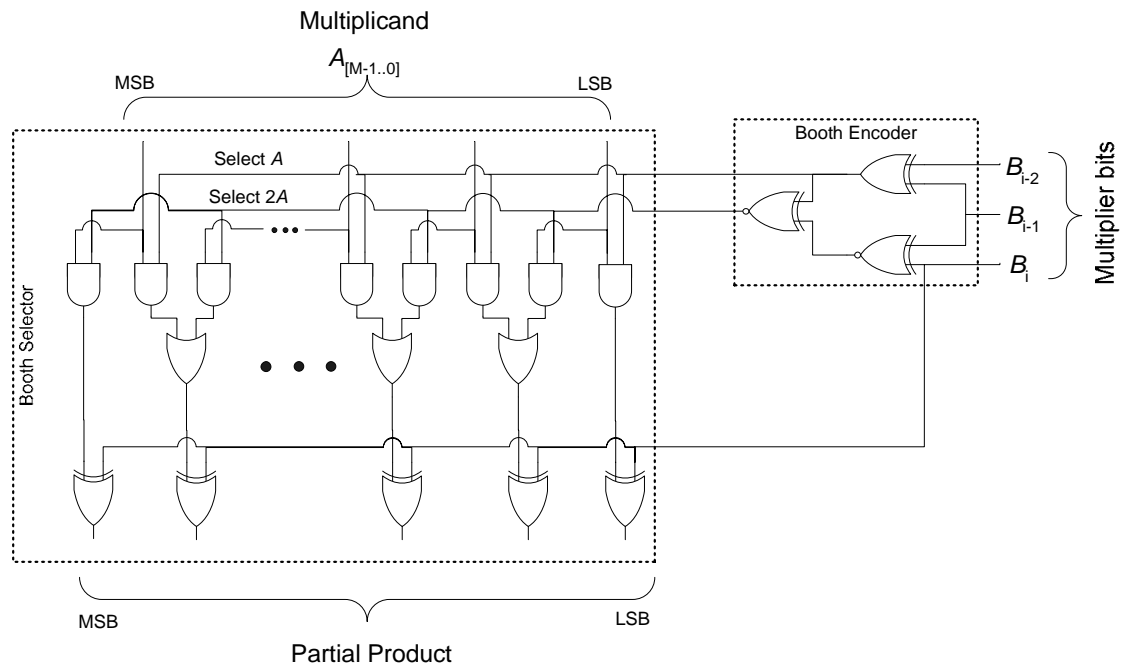


Figure 3.13: Bewick's implementation of the Booth encoder and decoder

### 3.5 Summary

This chapter discusses schemes for fast multiplication. Partial product generation using Modified Booth's algorithm and tree reduction using 4:2 compressors provides high performance. Both of these methods will be implemented for the multiply-accumulate unit in ASAP.

## Chapter 4

# Arithmetic Logic Unit Design and Implementation

An Arithmetic Logic Unit (ALU) is one of the main components in the datapath of a processor. It performs all of the logical operations and the simpler arithmetic operations (additions and subtractions). Since the ALU performs so much of the computation, a fast, efficient design is necessary. In this chapter the design and implementation of the ALU for the AsAP architecture is presented.

### 4.1 Instruction Set

The ALU performs 16-bit arithmetic on one or two input operands and creates one 16-bit output. There are also four branch registers that the ALU sets during each execution of an instruction. They are registers that hold information about whether the ALU result is negative ( $N$ ), zero ( $Z$ ), has a carry or borrow ( $C$ ), or has overflow ( $O$ ). Table 4.1 contains the instruction set of the ALU. The function  $F$  is a function of both 16-bit inputs  $A$  and  $B$ , and the carry register,  $C$ . The branch registers are also set on each operation, and an  $X$  in the column means the value for that particular register is set. Because of the real-time requirements, the saturating addition and subtraction operations are necessary for DSP architectures (Section 2.9, [26]).

| Instruction | $F(A,B,C)$                  | $N$ | $Z$ | $C$ | $O$ | Comments  |
|-------------|-----------------------------|-----|-----|-----|-----|---|
| No Function | $F = A$                     | $X$ | $X$ |     |     | Pass $A$ straight through   |
| NOT         | $F = \overline{A}$          | $X$ | $X$ |     |     | Invert $A$  |
| AND         | $F = A \& B$                | $X$ | $X$ |     |     | Bitwise AND   |
| NAND        | $F = \overline{A \& B}$     | $X$ | $X$ |     |     | Bitwise NAND  |
| OR          | $F = A   B$                 | $X$ | $X$ |     |     | Bitwise OR  |
| NOR         | $F = \overline{A   B}$      | $X$ | $X$ |     |     | Bitwise NOR   |
| XOR         | $F = A \oplus B$            | $X$ | $X$ |     |     | Bitwise XOR   |
| XNOR        | $F = \overline{A \oplus B}$ | $X$ | $X$ |     |     | Bitwise XNOR  |
| ANDWORD     | $F = \&A$                   |     | $X$ |     |     | AND all bits of $A$   |
| ORWORD      | $F =  A$                    |     | $X$ |     |     | OR all bits of $A$  |
| XORWORD     | $F = \oplus A$              |     | $X$ |     |     | XOR all bits of $A$   |
| BTRV        | $F = A'$                    | $X$ | $X$ |     |     | Bit Reverse, $A_0=A_{15}, A_1=A_{14}, \dots, A_{15}=A_0$                      |
| SHL         | $F = A \ll B$               | $X$ | $X$ |     |     | Logical left shift $A$ by $B$ places  |
| SHR         | $F = A \gg B$               | $X$ | $X$ |     |     | Logical right shift $A$ by $B$ places   |
| SRA         | $F = A \ggg B$              | $X$ | $X$ |     |     | Arithmetic right shift $A$ by $B$ places, sign extend $A$                     |
| ADD         | $F = A + B$                 | $X$ | $X$ | $X$ | $X$ | Add $A$ and $B$ , take lower 16 bits  |
| ADDC        | $F = A + B + C$             | $X$ | $X$ | $X$ | $X$ | Add $A, B, C$ , take lower 16 bits  |
| ADDS        | $F = A + B$                 | $X$ | $X$ | $X$ | $X$ | Add $A$ and $B$ , saturate result, take lower 16 bits                         |
| ADDCS       | $F = A + B + C$             | $X$ | $X$ | $X$ | $X$ | Add $A, B, C$ , saturate result, take lower 16 bits                           |
| ADDH        | $F = A + B$                 | $X$ | $X$ | $X$ | $X$ | Add $A$ and $B$ , take upper 16 bits  |
| ADDCH       | $F = A + B + C$             | $X$ | $X$ | $X$ | $X$ | Add $A, B, C$ , take upper 16 bits  |
| ADDINC      | $F = A + B + 1$             | $X$ | $X$ | $X$ | $X$ | Add $A, B, C$ , with a 1 forced into carry, take upper 16 bits                |
| SUB         | $F = A - B$                 | $X$ | $X$ | $X$ | $X$ | Subtract $B$ from $A$ , take lower 16 bits                                    |
| SUBC        | $F = A - B - C$             | $X$ | $X$ | $X$ | $X$ | Subtract $B, C$ from $A$ , take lower 16 bits                                 |
| SUBS        | $F = A - B$                 | $X$ | $X$ | $X$ | $X$ | Subtract $B$ from $A$ , saturate result, take lower 16 bits                   |
| SUBCS       | $F = A - B - C$             | $X$ | $X$ | $X$ | $X$ | Subtract $B, C$ from $A$ , saturate result, take lower 16 bits                |
| SUBH        | $F = A - B$                 | $X$ | $X$ | $X$ | $X$ | Subtract $B$ from $A$ , take upper 16 bits                                    |
| SUBCH       | $F = A - B - C$             | $X$ | $X$ | $X$ | $X$ | Subtract $B, C$ from $A$ , take upper 16 bits                                 |
| SUBDEC      | $F = A - B - 1$             | $X$ | $X$ | $X$ | $X$ | Subtract $B, C$ from $A$ , with a 0 forced into the carry, take upper 16 bits |

Table 4.1: Instruction set for the ALU in AsAP

| Control | In | Z |
|---------|----|---|
| 0       | 0  | 0 |
| 0       | 1  | 1 |
| 1       | 0  | 1 |
| 1       | 1  | 0 |

Table 4.2: Truth Table for XOR, used as a programmable inverter

## 4.2 Instruction Set Design and Implementation

Sections 4.2.1 – 4.2.4 discuss the designs to implement the instructions in Table 4.1.

### 4.2.1 Logic Operations Design and Implementation

This section covers the design of the “No Function”, NOT, AND, NAND, OR, NOR, XOR, and XNOR instructions in Table 4.1. Of these logic instructions, all of them have two sources,  $A$  and  $B$ , with the exception of the “No Function” and NOT instruction, which has one source,  $A$ . For these instructions we follow a bit-slice design where the logic for only one bit of the output needs to be created, and then arrayed out to the width of the ALU (in our case, 16 bits). The bit-slice contains three gates, the NOR, NAND, XOR, and a wire for the “No Function” operation. A selector before the output of the bit-slice selects the desired result, and each bit of the result is followed by an XOR gate that optionally inverts the result. An XOR gate can be used as a programmable inverter, where one input is a control signal that chooses whether the second input is inverted or not. Table 4.2 is the truth table for an XOR gate. Note how the input  $In$  is inverted when  $Control=1$ , and  $Z = In$  when  $Control=0$ . With this inverter, the additional instructions that can be implemented with this bit-slice are OR, AND, XNOR, and NOT. Figure 4.1 shows the array of 16 bit-slices. Figure 4.2 shows the 4-to-1 selector, which has four inputs, four control signals, and one output. The control signals to this selector will always have one and only one signal asserted, so as to have a low resistance path to the output. In general, an  $N$ -to-1 selector has  $N$  control signals.

### 4.2.2 Word Operations Design and Implementation

The three instructions ANDWORD, ORWORD, and XORWORD each have a single source and perform the equivalent of a 16-bit AND, OR, and XOR, respectively. Their functions are expressed in Equations 4.1, 4.2, and 4.3, and they return either a 1 or a 0 in the least significant bit of the result. The upper fifteen bits are always zero. In hardware, these functions do not fit the bit-slice architecture, and are comprised of a series of cascaded gates. Figure 4.3 shows the gate

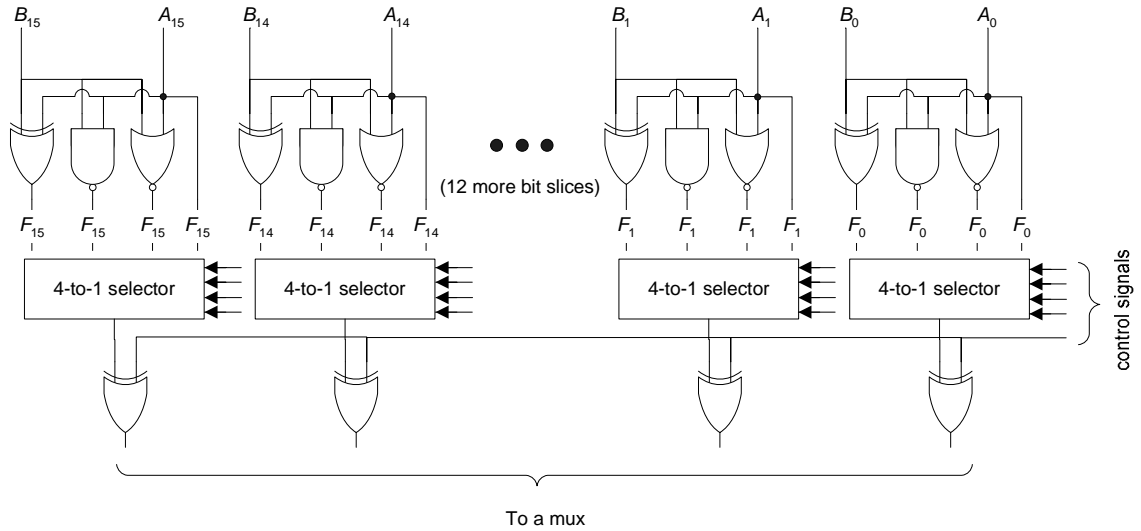


Figure 4.1: Array of 16 bit-slices for the ALU

schematics for these three functions. Each of the functions are implemented in four levels of logic. The XORWORD hardware uses 2-input XOR gates simply because this gate is commonly used in other circuits and is easy to instantiate copies in layout. The ORWORD hardware uses 2-input NOR gates because this limits the PMOS transistors in series to two. The ANDWORD hardware uses 4-input NAND gates because there is at most one series PMOS transistor in the pull-up network and there are four NMOS transistors in series in the pull-down network. For same sized transistors in the TSMC 0.18  $\mu\text{m}$  process, the mobility of an NMOS transistor is  $\sim 2.6$  times greater than that of a PMOS transistor, so four series transistors in the pulldown network each only need to be  $\sim 1.5$

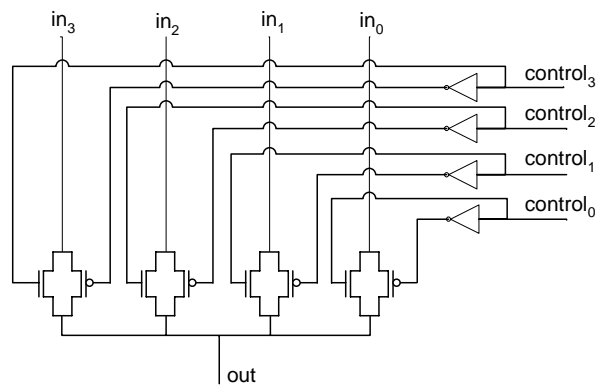


Figure 4.2: 4-to-1 selector using pass transmission gates

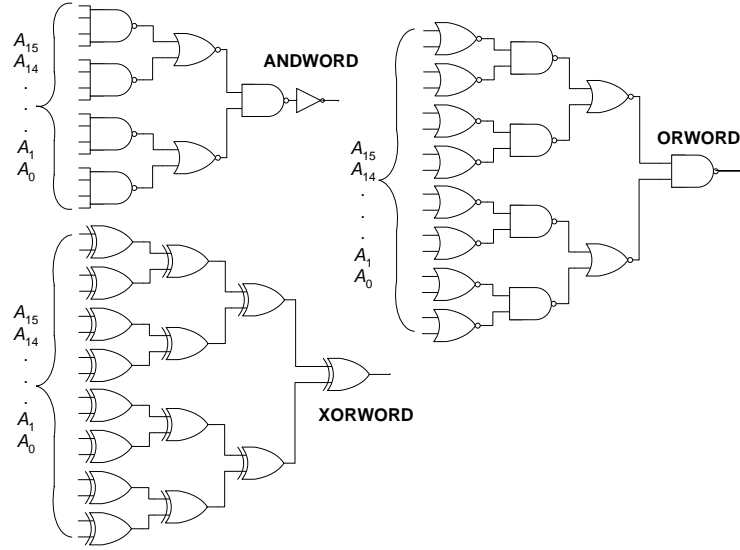


Figure 4.3: ANDWORD,ORWORD, and XORWORD gate schematics

times larger than the size of the one series PMOS transistor in the pullup network to provide the same resistance to output. The outputs of these three functions go to a 3-to-1 selector that selects the correct result.

$$ANDWORD = A_{15} \cdot A_{14} \cdot \dots \cdot A_1 \cdot A_0 \quad (4.1)$$

$$ORWORD = A_{15} + A_{14} + \dots + A_1 + A_0 \quad (4.2)$$

$$XORWORD = A_{15} \oplus A_{14} \oplus \dots \oplus A_1 \oplus A_0 \quad (4.3)$$

### 4.2.3 Bit Reverse and Shifter Design and Implementation

The BTRV, or bit-reverse instruction, takes the most significant bit of the source and makes it the least significant bit, the second most significant bit the second least significant bit, and so on. For the 16-bit case,  $F_{[15..0]} = A_0A_1A_2\dots A_{13}A_{14}A_{15}$  is the result of this instruction. In hardware, this simply requires a routing of 16 wires, and does not fit into the bit-slice architecture.

The shift instructions have two sources each:  $A$  and  $B$ , where the  $A$  input is the number to shift, and the lower four bits of  $B$  are the shift amount. The three shifters for the ALU are all implemented with a barrel shifter architecture [4]. A barrel shifter can shift an input any number of places in one operation (in a single cycle). It is possible to build a barrel shifter where the input passes through at most one transmission gate, but this would require as many control signals as there are input bits. By building a barrel shifter with multiple stages, the number of control signals

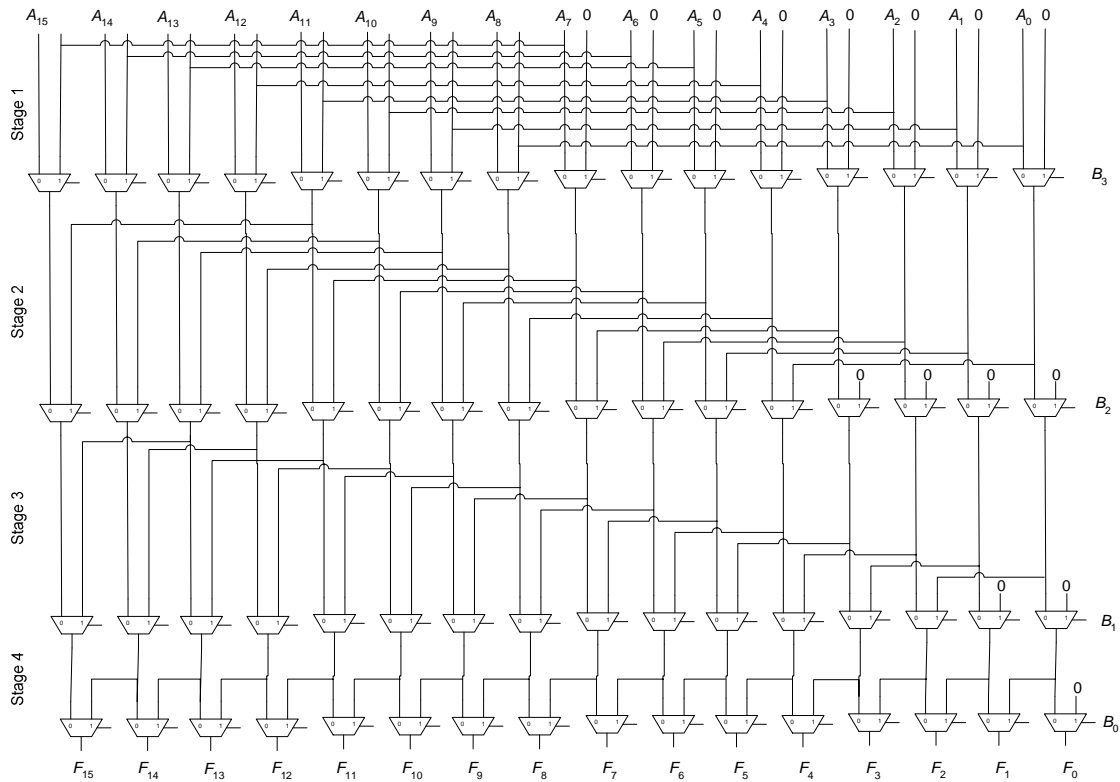


Figure 4.4: 16-bit logical left barrel shifter

are reduced by a factor of two for each additional stage, at the cost of another transmission gate to pass through. Using 2-to-1 muxes, each level in the shifter can shift a source by a power of two. For example, a one-level barrel shifter implemented with 2-to-1 muxes can shift a source up to one bit position. A two-level shifter can shift a source up to three bit positions, etc. In general, an  $N$ -level barrel shifter can shift a source up to  $2^N - 1$  places, when implemented with 2-to-1 muxes.

The SHL, or logical left shift, is implemented with four stages of muxes, for a maximum left shift of 15 places. When a left shift of one or more occurs, the bits shifted in from the right are zeroes. Figure 4.4 shows a schematic for the SHL instruction. The lower four bits of  $B$  are used as the select signals for the four rows of muxes. At each stage, a shift of either 8, 4, 2, or 1 place(s) occurs. These correspond to the control signals  $B_3, B_2, B_1, B_0$ , respectively. The input signals must pass through four transmission gates with four stages of muxes.

The SHR, or logical right shift, and the SRA, or arithmetic right shift, differ from each other in only one aspect. The difference is whether to fill in bits from the left with zeroes (SHL) or the sign bit (SRA). The underlying hardware is still designed with one right shifter, with an AND

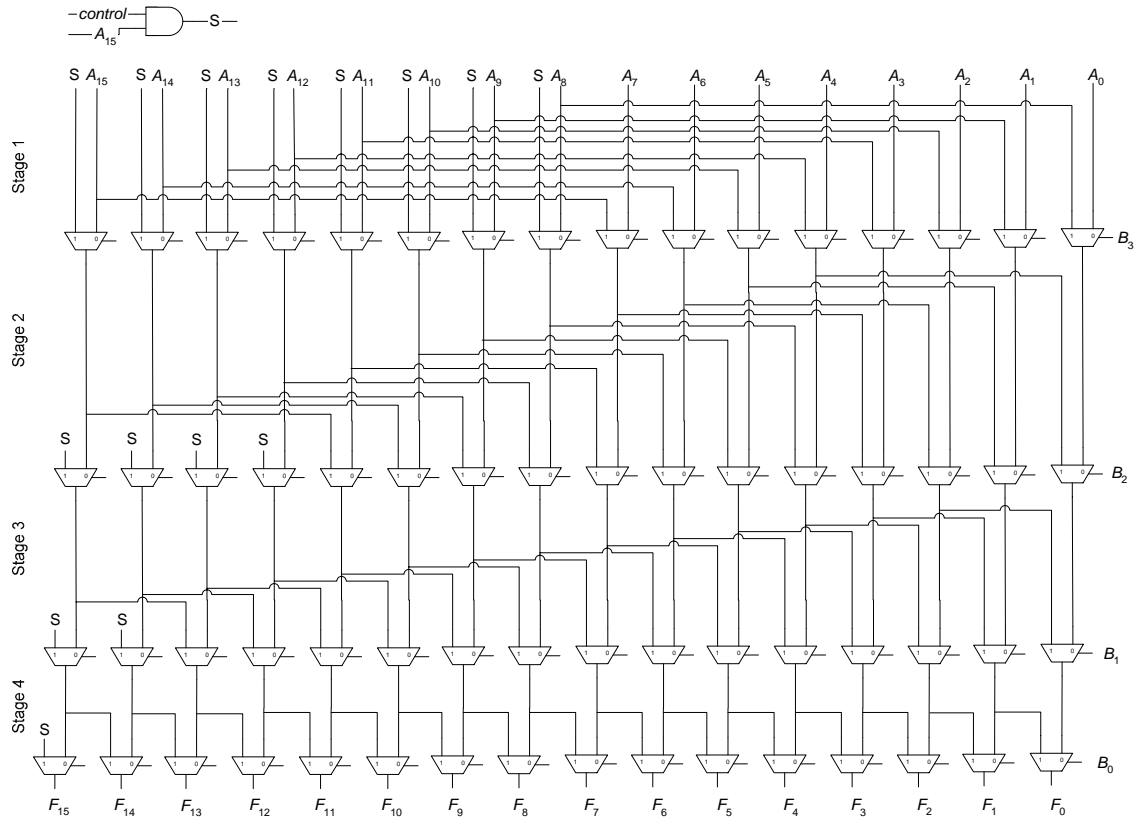


Figure 4.5: 16-bit logical and arithmetic right barrel shifter

gate and a control signal to choose between a zero or the sign bit. Figure 4.5 shows a schematic for the SHR and SRA instructions. When  $control=1$ , the sign bit is shifted in, when  $control=0$ , zero is shifted in. The control signals for the muxes are the same as for the SHL hardware, the lower four bits of  $B$ .

When implementing these shifters, it is difficult to include them in the bit-slice of Section 4.2.1 and automate the connections between the slices. Each input bit passes through four muxes before its final output, so each bit-slice would contain four muxes. However, the designer must still manually make the connections in these shifters after arraying out the slice. This problem arises because for the first three stages, the connections between each slice are not identical. Also, notice the vertical distance between stages in Figure 4.4. For the first stage, there is a height of eight wires between the inputs and the next row of muxes; for the second stage, there is a height of 12 wires between the rows of muxes; for the third stage, there is a height of 14 wires, and for the fourth stage, there is a height of one wire. In implementing the slice for the shifter, there must be enough room between the muxes to allow for the routing of these wires. That implies a vertical

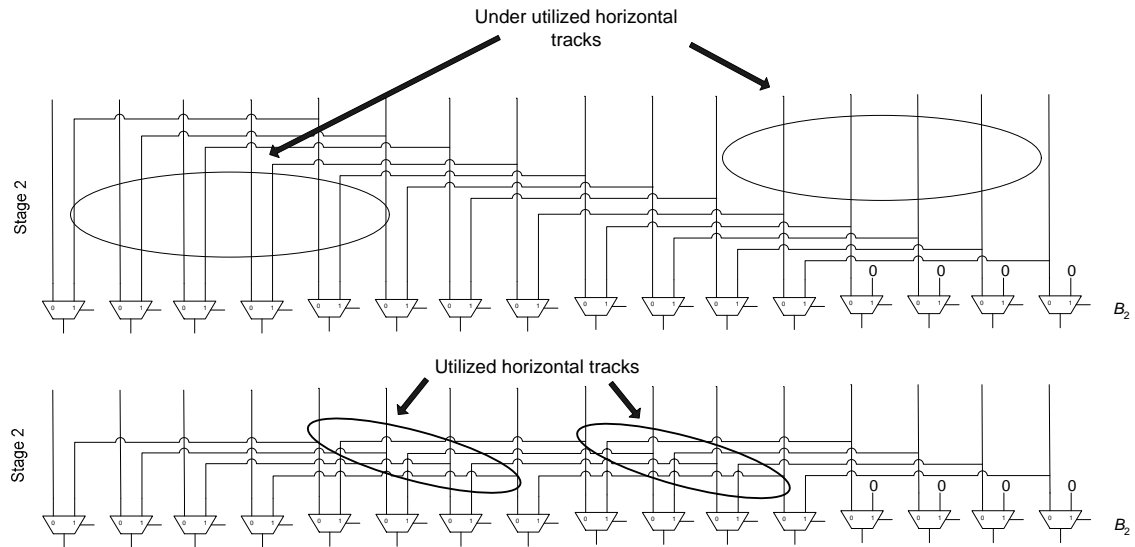


Figure 4.6: Illustration of fully utilized horizontal tracks in left shifter

spacing between the rows of muxes equivalent to the height of the wires between them. However, if we layout the slice of the shifter with a vertical spacing of 12 and 14 wires for the second and third stages, respectively, there would be a large waste of area. Upon closer examination of these two intermediate stages, there are horizontal tracks that are not being used, and wires could be routed. In the second stage, the minimum spacing between the row of muxes is the height of four wires. This is because at most four wires cross over other wires and need to be routed to another mux to the right. For the third stage the minimum height is two wires. Figure 4.6 shows the second stage of the SHL schematic with emphasis on the wasted area, and then the optimized Stage 2 which shows how the wires can be rerouted to utilize the previously empty horizontal tracks. Manual connection of the wires in layout is still required, but with this optimization the area of the shifter is reduced, and thus the speed of the shifter will also increase (due to shorter wire distance between stages). This same optimization is implemented in Stage 3, the only other stage where it is applicable.

### Bit Reverse and Shifter Implementation

The left and right shifters are symmetric with one exception: the right shifter needs to be able to shift in the sign bit for arithmetic shifts, whereas the left shifter always fills in zeroes. This symmetry allows us to take advantage of using just one shifter with selectors to select whether to perform a left shift or a right shift. The shifter for the ALU is implemented with a left shifter (Figure 4.4), and contains a row of 2-to-1 muxes both before the input and after the output that

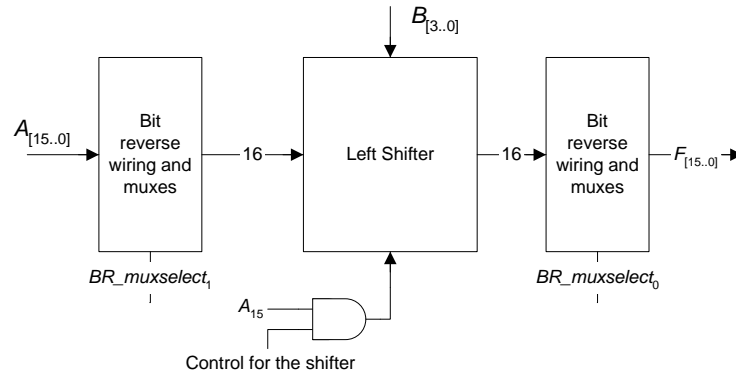


Figure 4.7: Block Diagram for left and right shifter, and bit reverse instruction

|                          | Implementation 1 | Our implementation |
|--------------------------|------------------|--------------------|
|                          | Area             | Area               |
| Left Shifter             | 5384             | 5384               |
| Right Shifter            | 5384             | 0                  |
| Bit reverse wiring       | 1208             | 0                  |
| Input bit reverse w/mux  | 0                | 1872               |
| Output bit reverse w/mux | 0                | 1872               |
| Buffer for right shifter | 1578             | 0                  |
| Buffer for left shifter  | 1578             | 1578               |
| Total Area               | 15132            | 10706              |

Table 4.3: Area comparison for Bit Reverse and Shifter Implementation: This table shows the area in  $\mu\text{m}^2$  for the shifters and bit reverse wiring. The Buffer is the area consumed by a row of buffers at the input of the shifters to drive through the shifters. For Implementation 1, these buffers do not have to be as large as the buffers in our implementation because it drives one fewer row of muxes, but this table is just for an estimate of the area savings. The result is an estimated area savings of 29%.

allow the option to bit reverse the input and the shifted result. To perform a left shift, the inputs are not bit reversed, and the output of the shifter is not bit reversed. To perform a right shift, the input is bit reversed, and so is the output of the shifter. With a mux to bit reverse the input, we observe that a secondary benefit to this mux is that the bit reverse instruction can be implemented with this shifter architecture. It can be done by bit reversing the input, shifting by zero through the left shifter, and then not bit reversing the output. Implementing each shifter and the bit reverse instruction individually would take eight rows of muxes and wiring for the bit reverse; but by implementing both shifters with just the left shifter and two rows of muxes, we save the two rows of muxes. This comes at the cost of additional wiring complexity because of the bit reverse of the input and output, but we are also implementing the bit reverse instruction with this shifter. Thus we trade off two rows of muxes for the wiring complexity of a bit reverse. Figure 4.7 shows the block diagram that

implements both shifters and the bit reverse instruction. Not shown in the figure is the row of buffers after the initial bit reverse and before the left shifter. The 2-to-1 muxes are implemented with pass transmission gates, and these buffers are meant to drive the signals through these next five transmission gate muxes (four in the shifter, and one in the bit reverse mux). To estimate the area savings, we obtain the area consumed by each shifter and bit-reverse wiring and present the results in Table 4.3. There is an estimated area savings of 29%.

#### 4.2.4 Adder/Subtractor Design and Implementation

The adder/subtractor implements all of the add and subtract instructions of Table 4.1. All of the arithmetic is signed and is done in the two's complement number system. This section describes in detail each of these instructions, and how they are implemented.

##### Addition instructions

The ADD instruction performs a 16-bit signed addition of the two sources  $A$  and  $B$ , produces a 17-bit intermediate result, and places the lower 16 bits at the output. To place the upper 16 bits at the output, the ADDH instruction is used. The ADDC instruction performs a 16-bit signed addition with the two sources  $A$ ,  $B$ , and the content of the carry register, and places the lower 16 bits at the output. To place the upper 16 bits at the output, use the ADDCH instruction. The ADDS instruction adds the two sources  $A$  and  $B$ , saturates the result if necessary, and places the lower 16 bits at the output. The ADDCS adds the two sources and the carry register, saturates the result if necessary, and places the lower 16 bits at the output. Saturation occurs when the addition of the two operands results in an overflow, and the result cannot represent the actual value of the result. For two positive numbers whose sum cannot be represented in 16 bits, saturating results in the largest possible 16-bit number. For two negative numbers whose sum cannot be represented in 16 bits, saturating results in the smallest possible 16-bit number. To determine when and how to saturate the result, refer to Table 4.4. The upper two bits of the 17-bit intermediate result are examined and determine the output for a saturate instruction. The ADDINC instruction adds the two sources and the carry register, but the value of the carry register will be set to 1. The upper 16 bits are used as a result, so the purpose of forcing a 1 in the carry is equivalent to rounding the addition by adding  $\frac{1}{2}$  an LSB.

| $F_{16}F_{15}$ | Status              | 16-bit result of F (HEX)   |
|----------------|---------------------|----------------------------|
| 00             | No saturation       | $F_{15}F_{14}\dots F_1F_0$ |
| 01             | Positive saturation | 7fff                       |
| 10             | Negative saturation | 8000                       |
| 11             | No saturation       | $F_{15}F_{14}\dots F_1F_0$ |

Table 4.4: Conditions for saturation and results

### Subtraction instructions

This section describes the way subtract instructions are implemented. With two's complement arithmetic, the negated value of a source can be found by inverting its bits and adding 1. For the subtractor unit in the ALU, we will simply use the adder, but invert one of the sources and add 1 when the instruction is a subtraction. This reuses the current hardware and saves area. The SUB instruction adds  $A$ ,  $\overline{B}$ , and the carry register with a 1 as its value. The output is the lower 16 bits of the 17-bit intermediate result. The SUBH instruction is similar to SUB except the output takes the upper 16 bits. The SUBC instruction adds  $A$ ,  $\overline{B}$ , and the inverted value of the carry register. This instruction takes the lower 16 bits as the result, and the SUBCH takes the upper 16 bits. The carry register, in relation to subtract instructions, is probably better referred to as the borrow register. If, during a SUBC instruction, the carry register is 1, it means that the previous subtract instruction borrowed a bit. The SUBS instruction adds  $A$ ,  $\overline{B}$ , and the carry register set to 1, and saturates using the conditions in Table 4.4. The SUBCS instruction adds  $A$ ,  $\overline{B}$ , and the inverted value of the carry register, and saturates the result. The SUBDEC instruction adds  $A$ ,  $\overline{B}$ , and the carry register set to 0.

For clarity, Table 4.5 summarizes this section by showing the actual arithmetic equations the adder/subtractor unit implements. The two sources are  $A$  and  $B$ , and the carry register is labeled  $C$  in the table. If a value is forced into the carry register, the Equation column in the table expresses that forced value. There are also columns that show whether there is saturation, and whether the lower or upper 16 bits are taken.

The block diagram for the adder/subtractor unit is shown in Figure 4.8. The output of the 16-bit signed adder block goes through a mux that selects the upper or lower 16 bits, and then another mux that chooses whether to saturate or not. Not shown in the figure are a row of buffers after the 16-bit signed adder and before the 16-bit hi/lo mux.

| Instruction | Equation                          | Saturate? | Lower 16 | Upper 16 |
|-------------|-----------------------------------|-----------|----------|----------|
| ADD         | $A + B + 0$                       | no        | X        |          |
| ADDC        | $A + B + C$                       | no        | X        |          |
| ADDS        | $A + B + 0$                       | yes       | X        |          |
| ADDCS       | $A + B + C$                       | yes       | X        |          |
| ADDH        | $A + B + 0$                       | no        |          | X        |
| ADDCH       | $A + B + C$                       | no        |          | X        |
| ADDINC      | $A + B + 1$                       | no        |          | X        |
| SUB         | $A + \overline{B} + 1$            | no        | X        |          |
| SUBC        | $A + \overline{B} + \overline{C}$ | no        | X        |          |
| SUBS        | $A + \overline{B} + 1$            | yes       | X        |          |
| SUBCS       | $A + \overline{B} + \overline{C}$ | yes       | X        |          |
| SUBH        | $A + \overline{B} + 1$            | no        |          | X        |
| SUBCH       | $A + \overline{B} + \overline{C}$ | no        |          | X        |
| SUBDEC      | $A + \overline{B} + 0$            | no        |          | X        |

Table 4.5: Add/Sub arithmetic equations and details

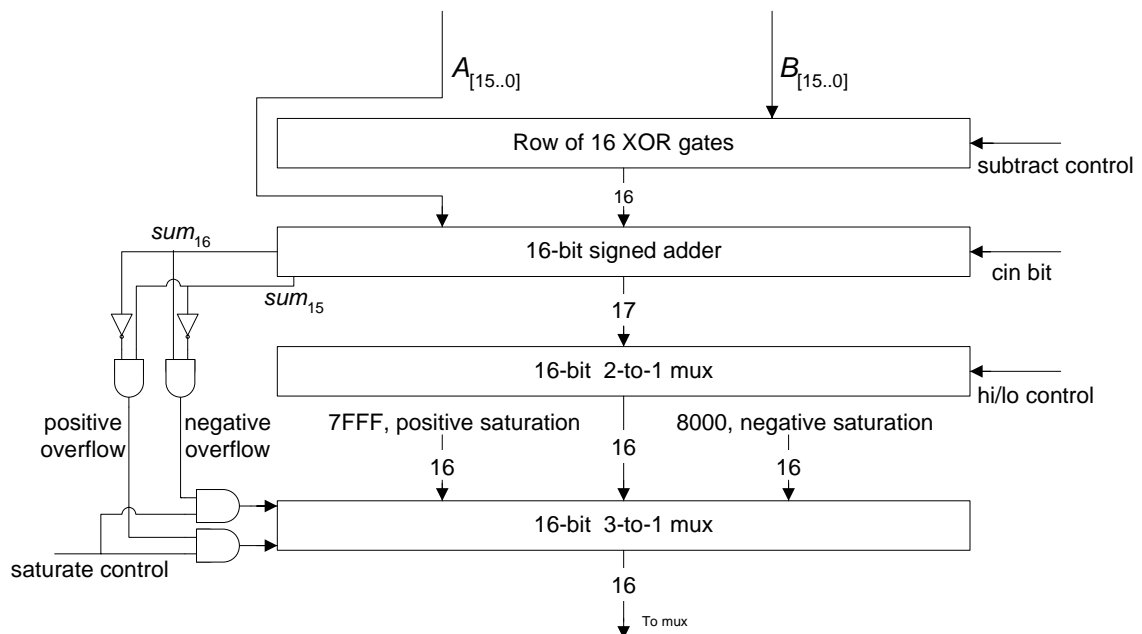


Figure 4.8: Adder/subtractor block diagram

### 4.2.5 Operational Code Selection

This section describes the operational codes (opcodes) for the instructions in the ALU. The opcode width for AsAP is six bits, but the ALU only requires the use of five bits. Choosing efficient opcodes for the ALU will simplify the decode logic. However, in AsAP the decode logic is already in a separate pipe stage than the ALU to speed up the operation of the ALU. Since the complexity of the ALU is much greater than the decode logic and it will be in a separate pipe stage, the speed of this decoder is not critical. The opcodes will be pre-decoded and the control signals to the ALU will be ready at the beginning of its pipe stage.

Table 4.6 is the truth table for the decode logic of the adder/subtractor unit. Its inputs are the lower five bits of the 6-bit opcode, and the outputs are the control signals for decoders. The outputs of these decoders are used as the control signals for the ALU. Table 4.7 is the truth table for the decode logic of the logic, word, and shift instructions. An X in the column is a “don’t care” for that particular control signal. When the instruction is an add/subtract instruction, all the control signals for the logic, word, and shift instructions are “don’t cares”, and vice versa. One control signal that is in the ALU and is not listed in either Table 4.6 or 4.7 is the one that selects between the output of the adder/subtractor or the logic, word, and shift instructions. The control signal is simply the 5<sup>th</sup> opcode bit  $P_4$ , where  $P_4=0$  selects the adder/subtractor result and  $P_4=1$  selects the logic, word, or shift result. In both these tables, the 6<sup>th</sup> opcode bit  $P_5$  is just listed as an input for completeness, but has no effect.

### 4.2.6 Decode Logic

Figure 4.9 shows a block diagram of the decode logic’s control signals and the ALU, in separate pipe stages. Equations 4.4 – 4.19 are the equations for the control signals of the decode logic, and Figure 4.10 shows the gate schematics for the decoder. Some of the control signals of Tables 4.6 and 4.7 have some of their signals fed into 2-to-4 decoders to further decode the control logic for the ALU. The motivation behind this is to reduce the computation done in the ALU pipe stage and ensure that the selectors have chosen a low resistance path to the output before the input arrives. The four branch registers are also set by the ALU during each clock cycle. The adder/subtractor unit sets the overflow (*bco*) and carry (*bcc*) register. These two registers use the upper two bits of the 17-bit intermediate result ( $temp_{[16..0]}$ ) calculated in each add/subtract operation. The negative (*bcn*) register uses the most significant bit of the ALU result and the zero (*bcz*) register performs a 16-bit NOR with the ALU result. The Boolean equations for these registers

| Opcode               | Instruction | Control Signals, generated by decode logic |              |                 |                   |       |       |
|----------------------|-------------|--|--------------|-----------------|-------------------|-------|-------|
| $P_5P_4P_3P_2P_1P_0$ |             | <i>subtract</i>                            | <i>hi/lo</i> | <i>saturate</i> | <i>carryvalue</i> | $s_1$ | $s_0$ |
| 0 0 0 0 0 0          | ADD         | 0  | 0            | 0               | 0                 | 0     | 0     |
| 0 0 0 0 0 1          | ADDC        | 0  | 0            | 0               | C                 | 1     | 1     |
| 0 0 0 0 1 0          | ADDS        | 0  | 0            | 1               | 0                 | 0     | 0     |
| 0 0 0 0 1 1          | ADDCS       | 0  | 0            | 1               | C                 | 1     | 1     |
| 0 0 0 1 0 0          | ADDH        | 0  | 1            | 0               | 0                 | 0     | 0     |
| 0 0 0 1 0 1          | ADDCH       | 0  | 1            | 0               | C                 | 1     | 1     |
| 0 0 0 1 1 0          | ADDINC      | 0  | 1            | 0               | 1                 | 0     | 1     |
| 0 0 0 1 1 1          | unused      | X  | X            | X               | X                 | X     | X     |
| 0 0 1 0 0 0          | SUB         | 1  | 0            | 0               | 1                 | 0     | 1     |
| 0 0 1 0 0 1          | SUBC        | 1  | 0            | 0               | $\overline{C}$    | 1     | 0     |
| 0 0 1 0 1 0          | SUBS        | 1  | 0            | 1               | 1                 | 0     | 1     |
| 0 0 1 0 1 1          | SUBCS       | 1  | 0            | 1               | $\overline{C}$    | 1     | 0     |
| 0 0 1 1 0 0          | SUBH        | 1  | 1            | 0               | 1                 | 0     | 1     |
| 0 0 1 1 0 1          | SUBCH       | 1  | 1            | 0               | $\overline{C}$    | 1     | 0     |
| 0 0 1 1 1 0          | SUBDEC      | 1  | 1            | 0               | 0                 | 0     | 0     |
| 0 0 1 1 1 1          | unused      | X  | X            | X               | X                 | X     | X     |

Table 4.6: Addition and subtraction opcodes and control signals: The inputs are the 6-bit opcode  $P_5, P_4, P_3, P_2, P_1, P_0$ . The subtract control signal is used for the row of programmable XORs before the 16-bit signed adder. The hi/lo control signal is the select line for a 16-bit 2-to-1 mux to select between the upper or lower 16 bits. The saturate control signal is to saturate the result if applicable.  $s_1$  and  $s_0$  are the select lines of a 3-to-1 mux to select what the carryin value will be to the adder/subtractor unit in Figure 4.8. It selects between 0, 1, the carry register ( $C$ ), or the inverted carry register value ( $\overline{C}$ ).

| Opcode               | Instruction | Control Signals, generated by decode logic |                |          |           |                |           |               |               |         |         |
|----------------------|-------------|--|----------------|----------|-----------|----------------|-----------|---------------|---------------|---------|---------|
| $P_5P_4P_3P_2P_1P_0$ |             | $logic\_mux_1$                             | $logic\_mux_0$ | $invert$ | $BRmux_1$ | $sign\_select$ | $BRmux_0$ | $word\_mux_1$ | $word\_mux_0$ | $lws_1$ | $lws_0$ |
| 0 1 0 0 0 0          | NOR         | 0  | 0              | 0        | X         | X              | X         | X             | X             | 0       | 0       |
| 0 1 0 0 0 1          | OR          | 0  | 0              | 1        | X         | X              | X         | X             | X             | 0       | 0       |
| 0 1 0 0 1 0          | NAND        | 0  | 1              | 0        | X         | X              | X         | X             | X             | 0       | 0       |
| 0 1 0 0 1 1          | AND         | 0  | 1              | 1        | X         | X              | X         | X             | X             | 0       | 0       |
| 0 1 0 1 0 0          | XOR         | 1  | 0              | 0        | X         | X              | X         | X             | X             | 0       | 0       |
| 0 1 0 1 0 1          | XNOR        | 1  | 0              | 1        | X         | X              | X         | X             | X             | 0       | 0       |
| 0 1 0 1 1 0          | PASS        | 1  | 1              | 0        | X         | X              | X         | X             | X             | 0       | 0       |
| 0 1 0 1 1 1          | NOT         | 1  | 1              | 1        | X         | X              | X         | X             | X             | 0       | 0       |
| 0 1 1 0 0 0          | ANDWORD     | X  | X              | X        | X         | X              | X         | 0             | 0             | 0       | 1       |
| 0 1 1 0 0 1          | ORWORD      | X  | X              | X        | X         | X              | X         | 0             | 1             | 0       | 1       |
| 0 1 1 0 1 0          | XORWORD     | X  | X              | X        | X         | X              | X         | 1             | 0             | 0       | 1       |
| 0 1 1 0 1 1          | unused      | X  | X              | X        | X         | X              | X         | X             | X             | X       | X       |
| 0 1 1 1 0 0          | SHL         | X  | X              | X        | 0         | 0              | 0         | X             | X             | 1       | 0       |
| 0 1 1 1 0 1          | SHR         | X  | X              | X        | 1         | 0              | 1         | X             | X             | 1       | 0       |
| 0 1 1 1 1 0          | SRA         | X  | X              | X        | 1         | 1              | 1         | X             | X             | 1       | 0       |
| 0 1 1 1 1 1          | BTRV        | X  | X              | X        | 1         | X              | 0         | X             | X             | 1       | 0       |

Table 4.7: Logic, Word, and Shift opcodes and control signals for the ALU: The  $logic\_mux_1$  and  $logic\_mux_0$  are control signals for a 2-to-4 decoder that selects between the NOR, NAND, XOR, or PASS functions, and the  $invert$  control signal is used to invert the signal if necessary (the PASS function is the equivalent of the “No Function” entry in Table 4.1). The  $BR\_mux_1$  and  $BR\_mux_0$  control signals are the select lines for the Bit Reverse muxes before and after the left shifter. The  $sign\_select$  is an input to the AND gate that selects whether to shift in 0 or  $A_{15}$  to the left shifter. The  $word\_mux_1$  and  $word\_mux_0$  are control signals to a 2-to-4 decoder that selects between the ANDWORD, ORWORD, or XORWORD instructions. The  $lws_1$  and  $lws_0$  are control signals to a 2-to-4 decoder which selects the result between a logic, word, or shift instruction.

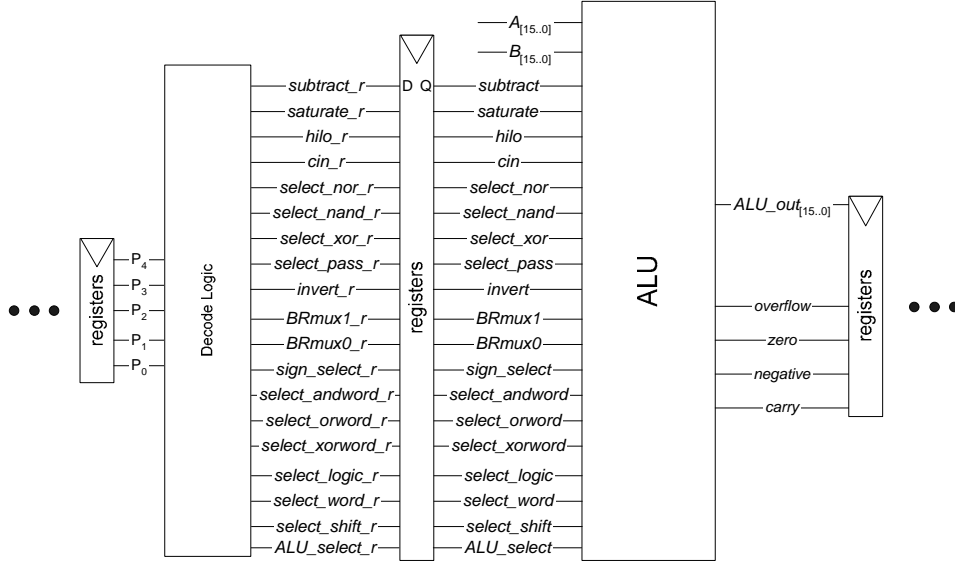


Figure 4.9: Decode Logic and ALU block diagram, in separate pipe stages: This diagram shows the lower 5 bits of the opcode as inputs to the Decode Logic, and the signals that the logic generates. These signals are registered and then fed into the ALU. The signals before the registers to the ALU have a ‘r’ appended to the end of the signal name, and those same signals come out of the registers without the ‘r’ appended.

are in Equations 4.20 – 4.23.

$$\text{subtract} = P_3 \quad (4.4)$$

$$\text{hi_lo} = P_2 \quad (4.5)$$

$$\text{saturate} = P_1(\overline{P_2} + P_0) \quad (4.6)$$

$$s_1 = P_0 \quad (4.7)$$

$$s_0 = \overline{P_3}(P_2P_1 + P_0) + P_3\overline{P_0}(\overline{P_1} + \overline{P_2}) \quad (4.8)$$

$$\text{logic\_mux}_1 = P_2 \quad (4.9)$$

$$\text{logic\_mux}_0 = P_1 \quad (4.10)$$

$$\text{invert} = P_0 \quad (4.11)$$

$$\text{BRmux}_1 = P_1 + P_0 \quad (4.12)$$

$$\text{BRmux}_0 = P_1 \oplus P_0 \quad (4.13)$$

$$\text{sign\_select} = P_1 \quad (4.14)$$

$$word\_mux_1 = P_1 \quad (4.15)$$

$$word\_mux_0 = P_0 \quad (4.16)$$

$$lws_1 = P_3P_2 \quad (4.17)$$

$$lws_0 = P_3\overline{P_2} \quad (4.18)$$

$$ALU\_select = P_4 \quad (4.19)$$

$$bco = temp_{16} \quad (4.20)$$

$$bcc = temp_{16} \oplus temp_{15} \quad (4.21)$$

$$bcn = ALU\_result_{15} \quad (4.22)$$

$$bcz = \overline{ALU\_result_{15} + ALU\_result_{14} + \dots + ALU\_result_1 + ALU\_result_0} \quad (4.23)$$

### 4.3 Layout and Performance

The block diagram for the ALU is in Figure 4.11. The layout occupies  $56,936 \mu\text{m}^2$ , with dimensions of  $290 \mu\text{m}$  by  $197 \mu\text{m}$  in TSMC  $0.18 \mu\text{m}$  technology using scalable SCMOS design rules. In the ALU, a total of five metal layers are used. The metal 5 layer is used only for the two sources,  $A$  and  $B$ , to the different ALU blocks. It is routed over the top of all the cells, which contain only metal layers 1 through 4. Thus a total of 32 metal 5 tracks are used and unavailable for use in the global layout. Figure 4.12 shows the layout for the ALU.

The layout is first extracted into an IRSIM netlist to test functionality (IRSIM reports a transistor count of 5326), and then extracted into an HSPICE netlist for the performance tests. The ALU can be clocked only as fast as its slowest instruction, therefore we need to run tests on every single instruction to determine the critical one. The input registers to the ALU have a clock-to-Q time plus a setup time of 250 ps (its schematic is shown in Figure 4.13 [6]). There are also large buffers after the input registers to drive the input signals to all the ALU blocks. To determine the speed of a particular instruction, we watch the  $ALU\_out$  signal, which is the 16-bit output bus before the 16-bit output register, as shown in Figure 4.11. We assert the control signals to the ALU that will perform the instruction we are testing, and also feed it the two input signals  $A$  and  $B$ , and the clock. The propagation delays are taken from the 50% point of the falling clock edge to the 50% point of the latest arriving output signal. Table 4.8 shows a test input for each instruction and the propagation delay. An additional 250 ps needs to be added to propagation delays in this table because that is the delay for the zero detect logic. The zero detect logic is implemented with

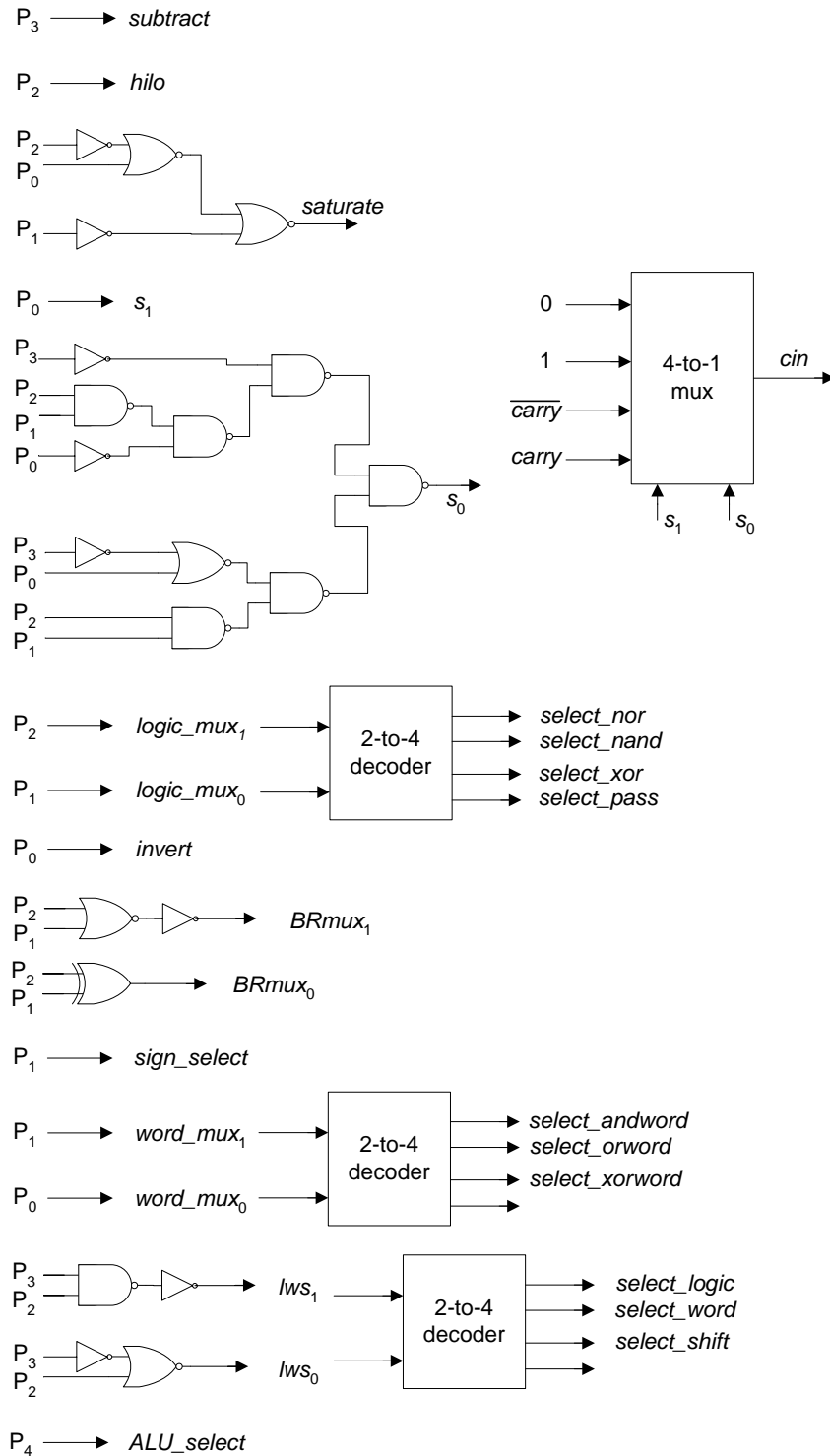


Figure 4.10: Gate Schematic for the Decode Logic for the ALU: Some of the control signals from Table 4.6 and 4.7 are fed into 2-to-4 decoders which generate the control signals for the ALU.

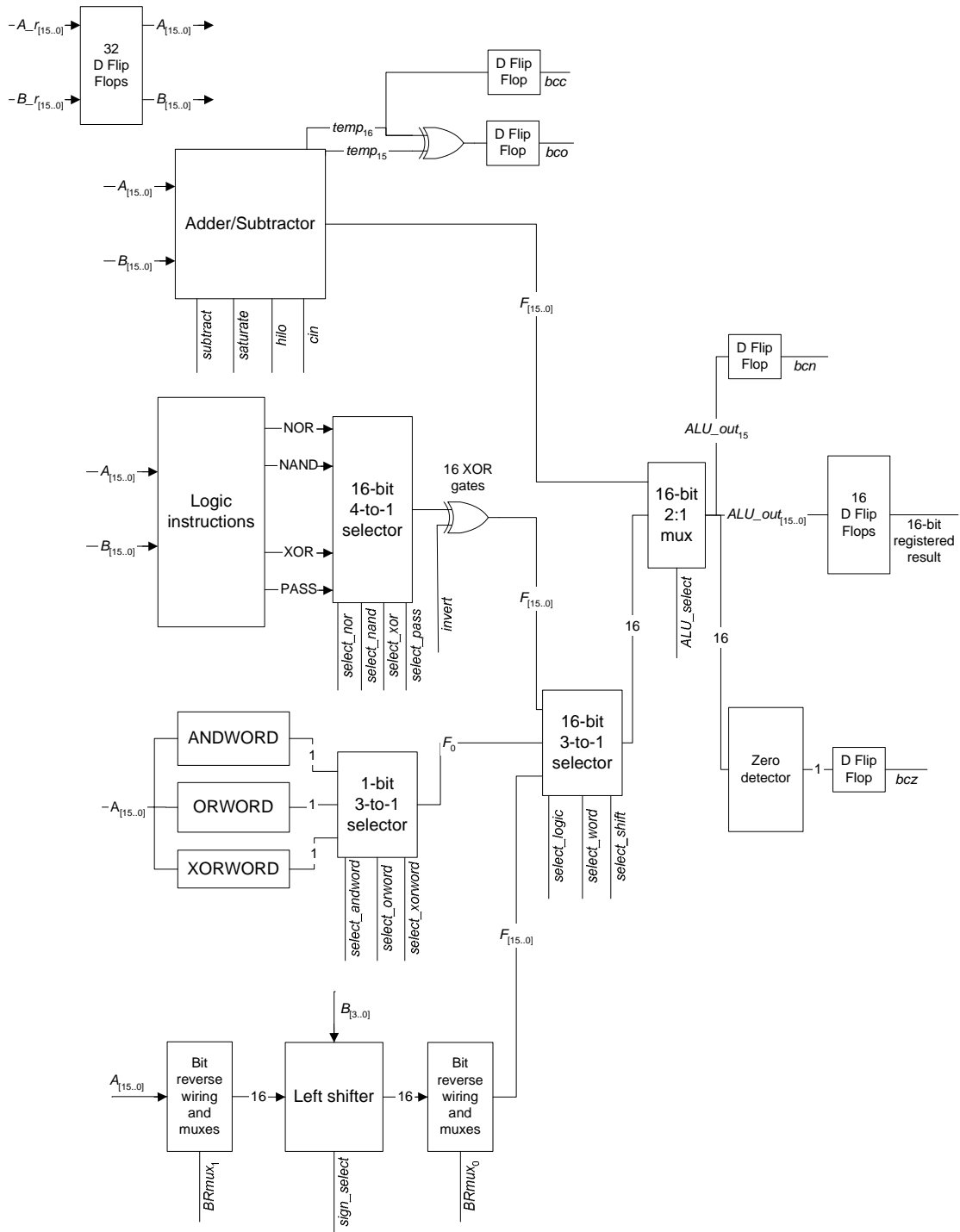


Figure 4.11: Block Diagram for ALU: This figure shows all the control signals the Decode Logic generates selecting the desired result from the ALU.

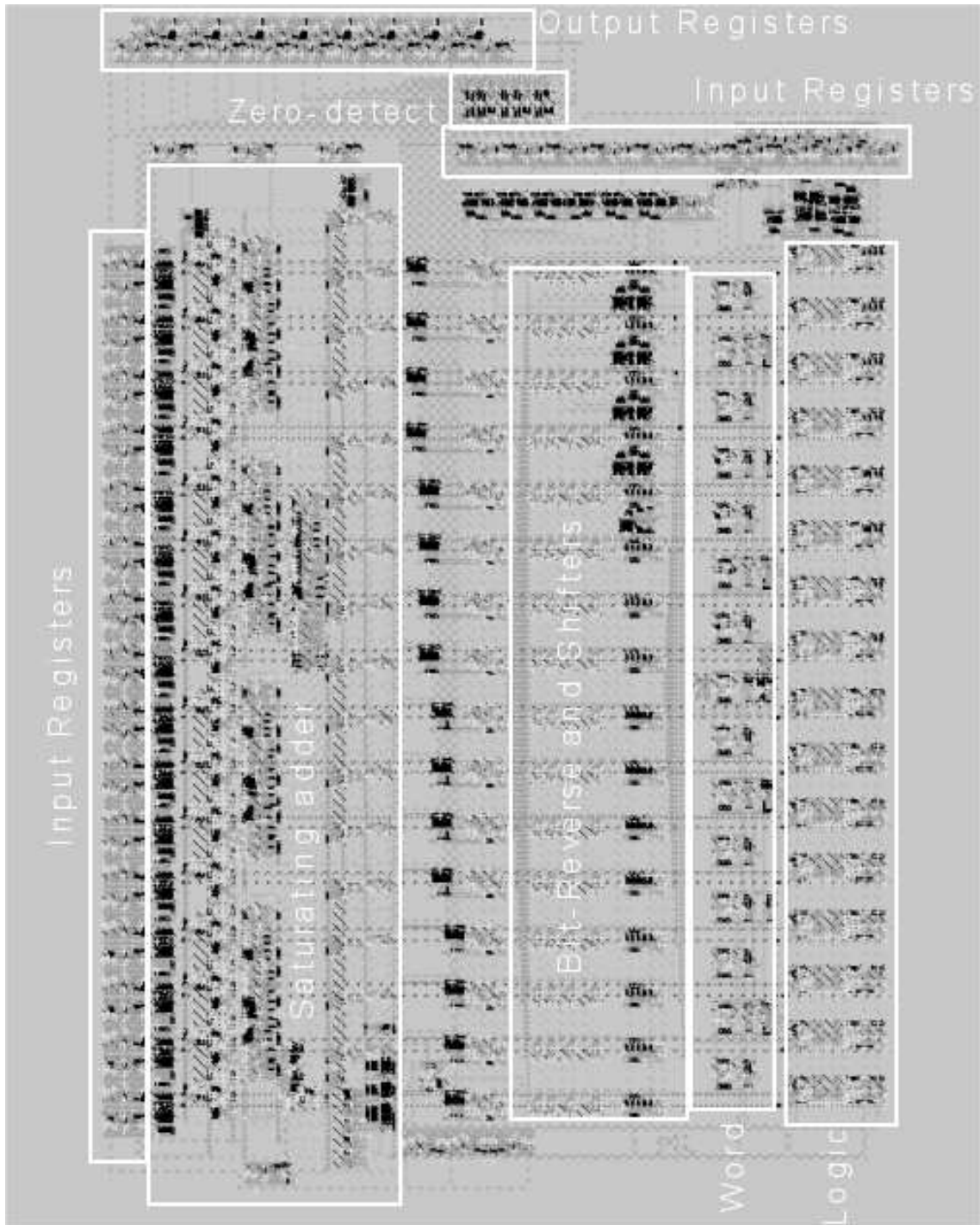


Figure 4.12: Final layout for the ALU

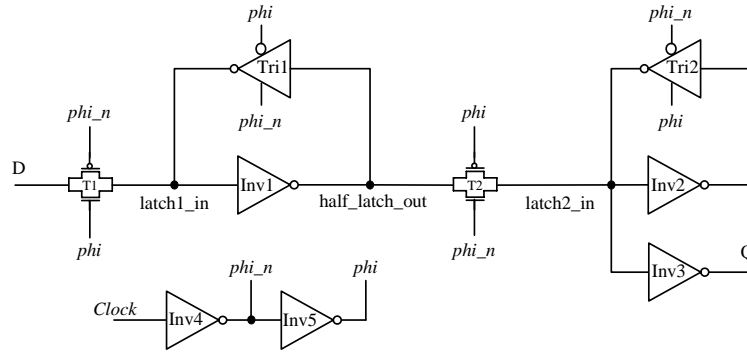


Figure 4.13: Schematic for negative edge triggered D flip-flop used in ALU [6]: The flip-flop uses transmission gates, inverters, and tri-stateable inverters. The input clock is locally buffered, and the unit has a clock-to-Q plus setup time of 250ps.

a 16-bit OR (Figure 4.3 followed by an inverter). The SUBS instruction delay plus the zero detect delay take 2.51 ns, allowing the ALU to be clocked at 398 MHz.

Upon further analyzing the SUBS instruction, we found that the output was initially just the result of a SUB instruction. However, after the saturate logic detects an overflow, a new result has to be calculated for the output. With the input vectors  $A=0x0001$  and  $B=0x8000$ , the initial result is  $0x8001$ . However, there is positive overflow in this case so that result needs to be changed to  $0x7FFF$ , which in this case requires almost all the bits to transition. This is what causes the critical propagation delay.

We also measure the energy consumed by the ALU. During every instruction, all operations are executed, but only one will be selected as the correct output. We measure the current while a SUBS instruction is being simulated because it causes almost all the output nodes to transition twice, and will probably draw the most current. The average current drawn is 16.1 mA over 4 ns, equaling a charge of 64.4 pC and 116 pJ of energy consumed. This results in a maximum power dissipation of 46.2 mW at the maximum operating frequency of 398 MHz.

## 4.4 Summary

This chapter presents the ALU design and implementation for the AsAP chip. It implements a total of 29 instructions, and can be clocked at a rate of 398 MHz.

| Instruction | Input <i>A</i> | Input <i>B</i> | <i>cin</i> | Prop. Delay    | Latest Arriving signal   |
|-------------|----------------|----------------|------------|----------------|--------------------------|
| PASS        | FFFF           | XXXX           | X          | 0.94 ns        | <i>out</i> <sub>15</sub> |
| NOT         | FFFF           | XXXX           | X          | 1.01 ns        | <i>out</i> <sub>3</sub>  |
| AND         | AAAA           | 5555           | X          | 1.11 ns        | <i>out</i> <sub>1</sub>  |
| NAND        | AAAA           | 5555           | X          | 1.04 ns        | <i>out</i> <sub>15</sub> |
| OR          | 0000           | 0000           | X          | 1.05 ns        | <i>out</i> <sub>15</sub> |
| NOR         | 0000           | 0000           | X          | 0.98 ns        | <i>out</i> <sub>15</sub> |
| XOR         | AAAA           | 5555           | X          | 1.19 ns        | <i>out</i> <sub>14</sub> |
| XNOR        | FFFF           | 5555           | X          | 1.21 ns        | <i>out</i> <sub>0</sub>  |
| ANDWORD     | FFFF           | XXXX           | X          | 0.88 ns        | <i>out</i> <sub>0</sub>  |
| ORWORD      | 0000           | XXXX           | X          | 0.93 ns        | <i>out</i> <sub>0</sub>  |
| XORWORD     | FFFE           | XXXX           | X          | 1.20 ns        | <i>out</i> <sub>0</sub>  |
| BTRV        | 0001           | 0000           | X          | 1.53 ns        | <i>out</i> <sub>15</sub> |
| SHL         | FFFF           | 000F           | X          | 1.60 ns        | <i>out</i> <sub>15</sub> |
| SHR         | FFFF           | 000F           | X          | 1.52 ns        | <i>out</i> <sub>0</sub>  |
| SRA         | FFFF           | 000F           | X          | 1.26 ns        | <i>out</i> <sub>8</sub>  |
| ADD         | FFFF           | 0000           | 0          | 1.76 ns        | <i>out</i> <sub>15</sub> |
| ADDC        | 7FFF           | 0000           | 1          | 1.61 ns        | <i>out</i> <sub>15</sub> |
| ADDS        | 7FFF           | 0001           | 0          | 2.12 ns        | <i>out</i> <sub>15</sub> |
| ADDCS       | 7FFF           | 0001           | 1          | 2.12 ns        | <i>out</i> <sub>15</sub> |
| ADDH        | FFFF           | 0000           | 0          | 1.74 ns        | <i>out</i> <sub>14</sub> |
| ADDCH       | FFFF           | 0000           | 1          | 1.57 ns        | <i>out</i> <sub>15</sub> |
| ADDINC      | FFFF           | 0000           | 1          | 1.57 ns        | <i>out</i> <sub>15</sub> |
| SUB         | FFFF           | 0000           | 1          | 1.49 ns        | <i>out</i> <sub>12</sub> |
| SUBC        | FFFF           | 0000           | 0          | 1.48 ns        | <i>out</i> <sub>12</sub> |
| SUBS        | 0001           | 8000           | 1          | <b>2.26 ns</b> | <i>out</i> <sub>15</sub> |
| SUBCS       | 0001           | 8000           | 0          | 2.25 ns        | <i>out</i> <sub>15</sub> |
| SUBH        | FFFF           | 0000           | 1          | 1.48 ns        | <i>out</i> <sub>11</sub> |
| SUBCH       | FFFF           | 0000           | 0          | 1.46 ns        | <i>out</i> <sub>15</sub> |
| SUBDEC      | FFFF           | 0000           | 0          | 1.48 ns        | <i>out</i> <sub>11</sub> |

Table 4.8: ALU performance: This table lists a test for each instruction the ALU can execute. It shows that a subtraction with saturation is the critical path. The propagation delays for these signals are measured from the 50% of the falling clock edge to the 50% input of the output registers, which have near zero setup time. An additional 250 ps needs to be added to the critical path for the zero detect logic.

## Chapter 5

# Multiply-Accumulate Unit Design and Implementation

The Multiply-Accumulate (MAC) unit performs the Multiply instruction and the MAC instruction, which are essential for all DSP processors. In order to achieve high performance, the MAC unit is pipelined into three stages. This chapter discusses the design and implementation of the multiply-accumulate unit for the AsAP.

### 5.1 Instruction Set

The MAC unit takes two 16-bit input operands and creates two 16-bit results. One output, *mult\_out* is the result of a Multiply or MAC instruction and the other output, *acc16*, is the lowest 16 bits of the 40-bit accumulator. The 40-bit accumulator accumulates the results of each MAC instruction. The MAC instructions also write to *mult\_out*, and the programmer has the option of using either *mult\_out* or *acc16*. The Multiply instructions only write to *mult\_out*, and the accumulator value is preserved. The instruction set for the MAC unit is shown in Table 5.1.

### 5.2 Instruction Set Design and Implementation

The MAC unit is divided into three pipe stages so it has the ability to be clocked at a faster rate. The latency is three cycles for a MAC or multiply instruction, but the throughput is one operation per cycle. Figure 5.1 shows a block diagram of the MAC unit. The first pipe stage

| Instruction | $mult\_out(A, B)$ | $acc16(A, B)$          | Comments                        |
|-------------|-------------------|------------------------|---------------------------------|
| MULTL       | $A \cdot B$       | previous value         | $mult\_out$ takes lower 16 bits |
| MULTH       | $A \cdot B$       | previous value         | $mult\_out$ takes upper 16 bits |
| MACL        | $A \cdot B$       | $A \cdot B + acc40$    | $mult\_out$ takes lower 16 bits |
| MACH        | $A \cdot B$       | $A \cdot B + acc40$    | $mult\_out$ takes upper 16 bits |
| MACCL       | $A \cdot B$       | $A \cdot B$            | $mult\_out$ takes lower 16 bits |
| MACCH       | $A \cdot B$       | $A \cdot B$            | $mult\_out$ takes upper 16 bits |
| ACCSHR      | invalid           | $acc40 \gg A_{[4..0]}$ | right shift accumulator         |

Table 5.1: Instruction Set for the MAC unit in AsAP:  $mult\_out$  is a 16-bit register which can be the result of either a multiply or MAC instruction.  $acc40$  is the output of the entire accumulator, whereas  $acc16$  is the lowest 16 bits of  $acc40$ . The multiply instructions (MULTL, MULTH) write to the  $mult\_out$  register, but does not write to the accumulator. The MAC instructions (MACL, MACH) write to the  $mult\_out$  register and to the accumulator. The MACC instructions (MACCL, MACCH) write to the  $mult\_out$  register and overwrite the value of the accumulator with the product of  $A$  and  $B$ . The ACCSHR instruction has a maximum right shift of 16 places, with the lower five bits of  $A$  as the shift amount. It writes to the accumulator.

contains the Booth encoding, generation of partial products (Booth decoding), and two rows of 4:2 compressors to begin accumulation of the partial products. The second pipe stage continues accumulation of the partial products with a row of 4:2 compressors and a row of half adders. After this pipe stage the partial product tree is reduced to two rows of 32 bits. The third and final pipe stage contains a 40-bit carry propagate adder (CPA) to complete the MAC or multiply instruction, the 40-bit accumulator, and the accumulator shifter. This stage is the critical pipeline stage of the MAC unit.

### 5.2.1 Partial Product Generation

The first step in a multiplication is to generate the partial product bits. In this implementation, modified Booth encoding is chosen because of the simple task of generating the multiplicand and twice the multiplicand for the partial products. The benefits of Booth encoding in multipliers have been questioned because of the efficiency of 4:2 counters versus the complex logic of Booth encoders and selectors [27]. In that work, the authors show that 4:2 compressors achieve faster reduction of the partial product tree than Booth encoding at the expense of more area. We try to achieve an area/speed balance, and the partial product generation for this unit is not in the critical pipe stage. For these reasons we implement the partial product tree with Booth encoders because it saves area and the speed will be faster than the critical pipeline stage.

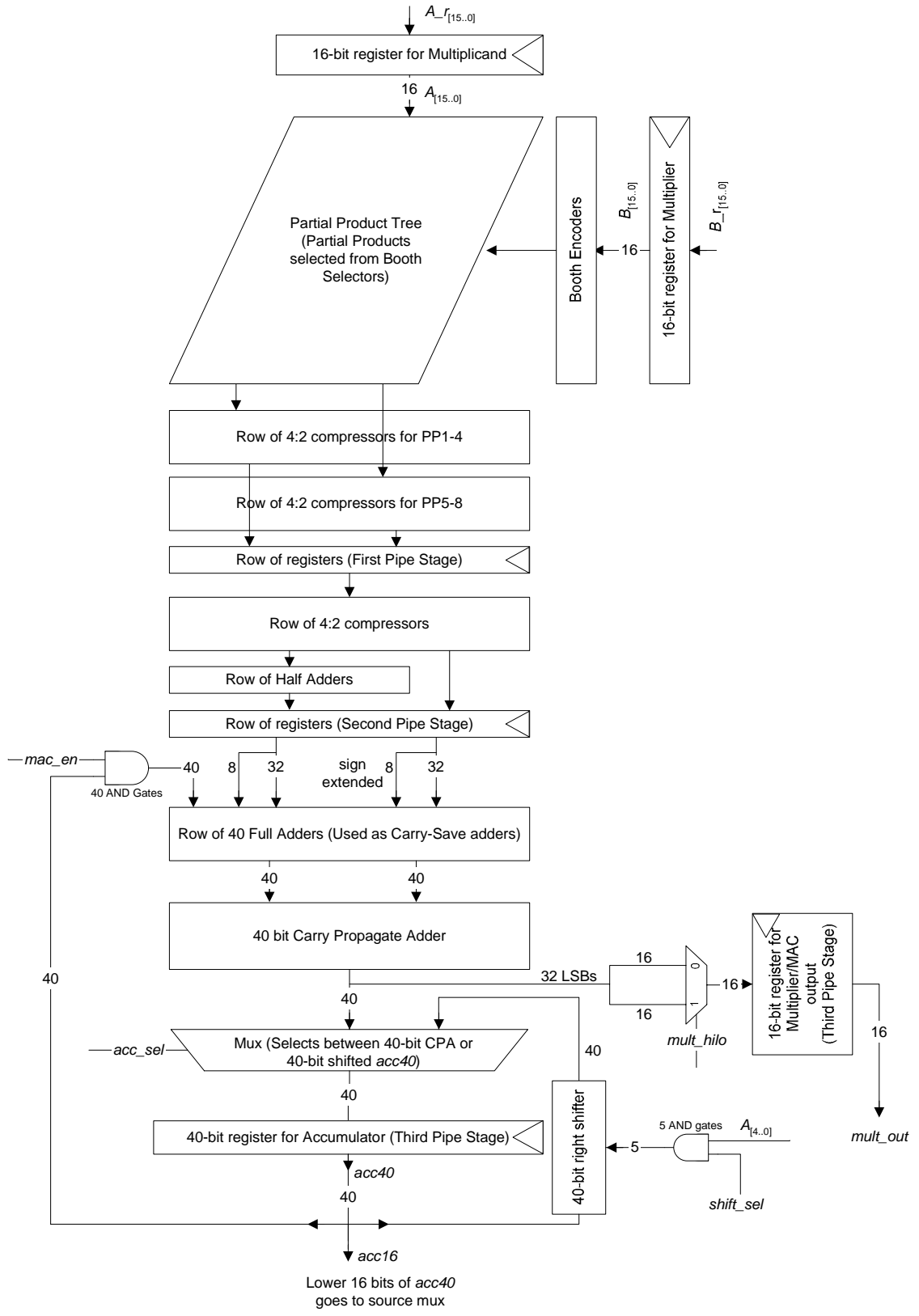


Figure 5.1: Block diagram for MAC unit

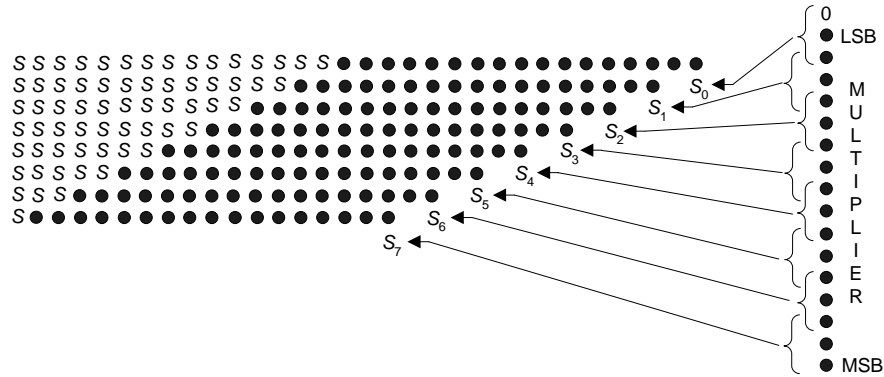


Figure 5.2: Partial Product Tree with sign extension: The bits in the least significant position of each partial product ( $S$  with a subscript) are used to negate the partial product if necessary. The bits labeled  $S$  in the most significant positions are the sign extension of the most significant bit of their respective partial product.

### Sign extension

The multiplicand determines the width of the partial product tree, and the multiplier determines the height. Using Modified-Booth encoding for these two signed numbers results in a tree that is nine rows high. The width of the tree is 32 bits wide because the final result of a  $16 \times 16$ -bit multiplication is 32 bits. Also, each partial product generated needs to be sign extended, and also needs to add an additional bit in the least significant position for negative partial products (the “invert and add 1” for negative numbers in the two’s complement number system). There are a total of eight partial products, and each one is 17 bits wide (before sign extension). The additional bit in the least significant position of each partial product increases the height of the tree to nine. Figure 5.2 shows the partial product tree with sign extension.

Although necessary, sign extension in the partial product rows is inefficient because it requires additional hardware to add the sign bits. Also, sign extension presents a large load to the most significant bit of each partial product row, because the most significant bit needs to be replicated many times. This would significantly slow down the partial product generation step. In order to alleviate this sign extension problem, we use a method that reduces the sign extension bits to a constant, and instead add this constant to the partial product tree during every multiplication operation. Bewick’s appendix [25] has a thorough explanation for this method, and goes through an example for a  $16 \times 16$ -bit multiplier. This method is also discussed by other authors ([28] and Section 6.3, [9]). Figure 5.3 shows and explains the steps taken to reduce the number of bits to add in the partial product tree for the  $16 \times 16$ -bit signed multiplication needed in this MAC unit. After

| Inputs |       |       | Outputs   |            |           |        |
|--------|-------|-------|-----------|------------|-----------|--------|
| $m_2$  | $m_1$ | $m_0$ | $selectm$ | $select2m$ | $select0$ | $sign$ |
| 0      | 0     | 0     | 0         | 0          | 1         | 0      |
| 0      | 0     | 1     | 1         | 0          | 0         | 0      |
| 0      | 1     | 0     | 1         | 0          | 0         | 0      |
| 0      | 1     | 1     | 0         | 1          | 0         | 0      |
| 1      | 0     | 0     | 0         | 1          | 0         | 1      |
| 1      | 0     | 1     | 1         | 0          | 0         | 1      |
| 1      | 1     | 0     | 1         | 0          | 0         | 1      |
| 1      | 1     | 1     | 0         | 0          | 1         | 1      |

Table 5.2: Truth table for the Booth encoder.  $m_2$ ,  $m_1$ , and  $m_0$  are three bits of the multiplier, and four control signals are generated to select the correct partial product.

the fourth step the partial product bits are ready for accumulation.

### Booth Encoders and Partial Product Selectors

This section describes how to generate the partial product tree in the fourth step of Figure 5.3. Each partial product row is generated with a Booth encoder and a partial product row selector (Booth selector). Its block diagram is shown in Figure 5.4. The truth table for the Booth encoder is in Table 5.2, its equations expressed in Equations 5.1 – 5.4, and its gate schematic is in Figure 5.5. The encoder has three inputs and produces four outputs, which are select signals for the partial product selector. The select signals from the Booth encoder have drivers at the end to drive the long wire and large amount of logic in the partial product row selector.

$$selectm = m_1 \oplus m_0 \quad (5.1)$$

$$select2m = \overline{m_2} \cdot m_1 \cdot m_0 + m_2 \cdot \overline{m_1} \cdot \overline{m_0} \quad (5.2)$$

$$select0 = \overline{m_2} \cdot \overline{m_1} \cdot \overline{m_0} + m_2 \cdot m_1 \cdot m_0 \quad (5.3)$$

$$sign = m_2 \quad (5.4)$$

The partial product row selector contains 17 partial product bit selectors and an inverter at the end to invert the sign bit. The truth table for one partial product bit is in Table 5.3, its equation in Equation 5.5, and the gate schematic for one partial product bit is in Figure 5.6.

$$pp\ bit = sign \oplus (selectm \cdot mcand + select2m \cdot (2 \cdot mcand) + select0 \cdot Gnd) \quad (5.5)$$

It takes as inputs the four output signals from the Booth encoder and two bits of the multiplicand, and it outputs one bit. The logic for this cell is a 3-to-1 selector that chooses between the multiplicand ( $mcand$ ), twice the multiplicand ( $2 \cdot mcand$ ), or zero. There is also the option to invert the result

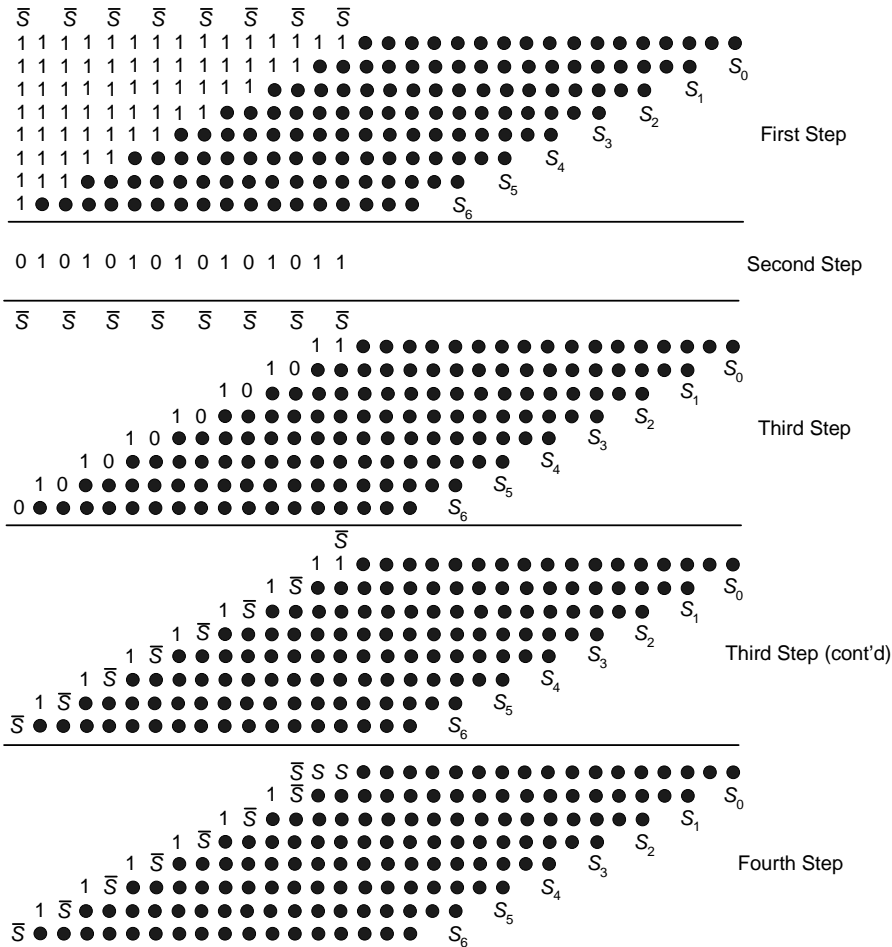


Figure 5.3: Reducing sign extension in a partial product tree: Each step in this figure has a dot diagram of the partial product tree with the sign extension bits. The  $S_7$  bit is omitted to reduce the height of the tree and will be added at a later pipe stage. The first step involves assuming all the partial products are negative, and sign extending with 1's. If the partial product turns out to be positive, the string of ones can be converted back to leading 0's by adding a 1 in the least significant position. This is achieved by adding the inverted sign bit,  $\bar{S}$ .  $\bar{S} = 0$  for a negative partial product, thus keeping the string of ones; and  $\bar{S} = 1$  for a positive partial product, thus turning the extension into a string of zeroes. The second step adds all the sign extension bits into a single constant. In the figure this step omits the other dots in the partial product tree. The third step fills in this constant with the rest of the bits. In the hardware, these 1's in the constant will be hardwired to  $V_{dd}$ . The fourth step reduces the height of the tree by absorbing the  $\bar{S}$  into the first partial product. After the fourth step, the partial product tree is ready to be accumulated.

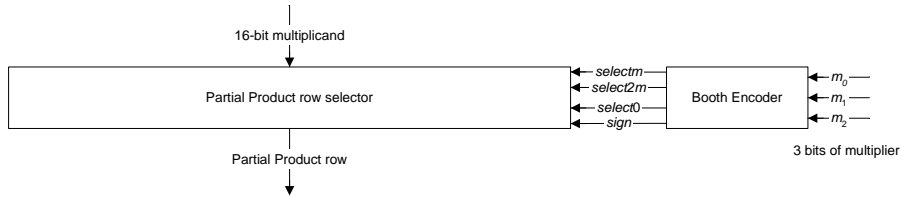


Figure 5.4: Block diagram for Booth encoder and partial product selector

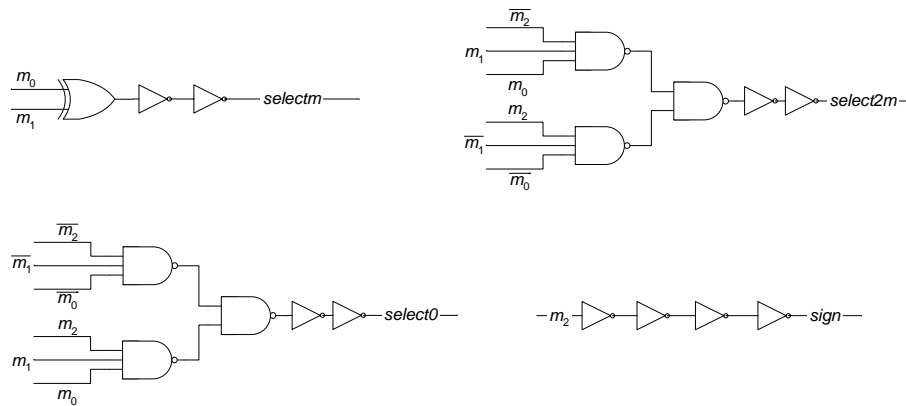


Figure 5.5: Gate schematic for Booth encoder

that this selector chooses. The XOR gate in this cell is used as a programmable inverter, and is implemented with an XNOR gate followed by an inverter so as to buffer the output signal and give the cell driving power. The *sign* control signal either inverts the partial product bit or not. In our implementation of the MAC unit, the partial product bit selector is arrayed horizontally 17 times to build a block called a partial product row selector, and the detailed block diagram is in Figure 5.7. The largest result that this row can output is twice the multiplicand (at most a left shift of one), hence the cascade of 17 cells. Notice in Table 5.3 that there are four inputs (for a maximum of 16 unique combinations), but there are only eight entries. This is because the assertion of a select signal is mutually exclusive amongst the other two (as described by the Booth encoder). Only *selectm*, *select2m* or *select0* can be asserted at any time.

The design of this Booth encoder is based on the work done on a 16-bit multiplier for a baseband transmitter [29]. In that paper, Zeydel *et al.* design a Booth encoder with one-hot encoding and the partial product selector as a mux with six transmission gates. The wiring complexity for their partial product cell is large, with 18 wires running through the cell. This is because there

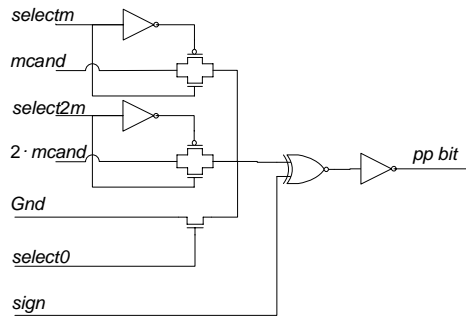


Figure 5.6: Gate schematic for a partial product bit selector

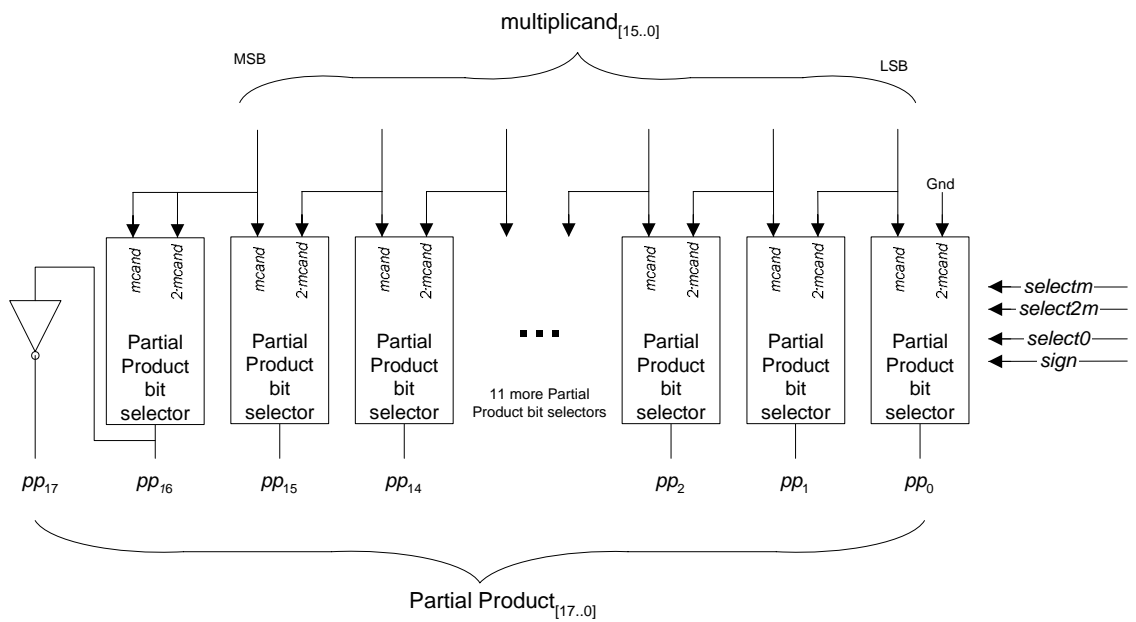


Figure 5.7: Gate schematic for the partial product row selector

| Inputs         |                 |                |             | Output                     |
|----------------|-----------------|----------------|-------------|----------------------------|
| <i>selectm</i> | <i>select2m</i> | <i>select0</i> | <i>sign</i> | <i>pp bit</i>              |
| 0              | 0               | 1              | 0           | 0                          |
| 0              | 1               | 0              | 0           | $2 \cdot mcand$            |
| 1              | 0               | 0              | 0           | $mcand$                    |
| 0              | 0               | 1              | 1           | 1                          |
| 0              | 1               | 0              | 1           | $\overline{2 \cdot mcand}$ |
| 1              | 0               | 0              | 1           | $\overline{mcand}$         |

Table 5.3: Truth table for the partial product bit selector

are six inputs to select from ( $Gnd$ , multiplicand, 2-multiplicand, and their complements), and two select signals for each of these inputs. In our implementation we reduce the wiring complexity of the cell by locally inverting all signals that need to be complemented. The implementation of our partial product bit selector has eight wires running through the cell ( $V_{dd}$ ,  $Gnd$ ,  $mcand$ ,  $2 \cdot mcand$ , and the four select signals).

### 5.2.2 Partial Product Accumulation

After the partial products are generated, the next step is to accumulate them. The goal is to reduce the tree to carry-save format, which is two rows of bits. When the tree is in this format, a carry propagate adder (as discussed in Chapter 2) produces the final sum.

We primarily use 4:2 compressors (as discussed in Section 3.3.3) because of their speed and regularity in layout. The partial product tree in the fourth step of Figure 5.3 is reduced with two rows of 4:2 compressors (which can cover a tree of height 8). The  $S_7$  bit makes the height of the tree 9 bits high, and two rows of 4:2's would not be sufficient, so we delay the addition of this bit to a later pipe stage. These two rows of 4:2's, the Booth encoder, and partial product row selectors are in the first pipe stage. Its outputs are fed into registers for the next pipe stage. There are a total of 96 registers at the end of the first pipe stage (92 registers for the outputs of the 4:2 compressors, 2 registers for the  $\bar{S}$  sign extension bits, and 1 register each for  $S_3$  and  $S_7$ ).

The second pipe stage continues the tree reduction. The tree now has a height of five rows (four rows from the outputs of the 4:2 compressors from the first pipe stage, and one row for the  $S_7$  bit). One row of 26 4:2 compressors is used for the first four rows of bits, which compresses into two rows of bits. It is possible to use a row of 32 4:2 registers in this pipe stage, but we observe that the bits in the least significant portion of this pipe stage are already in carry-save format. After the 4:2 compressor reduction, there are now three rows in the tree with the  $S_7$  bit. The outputs of the 26

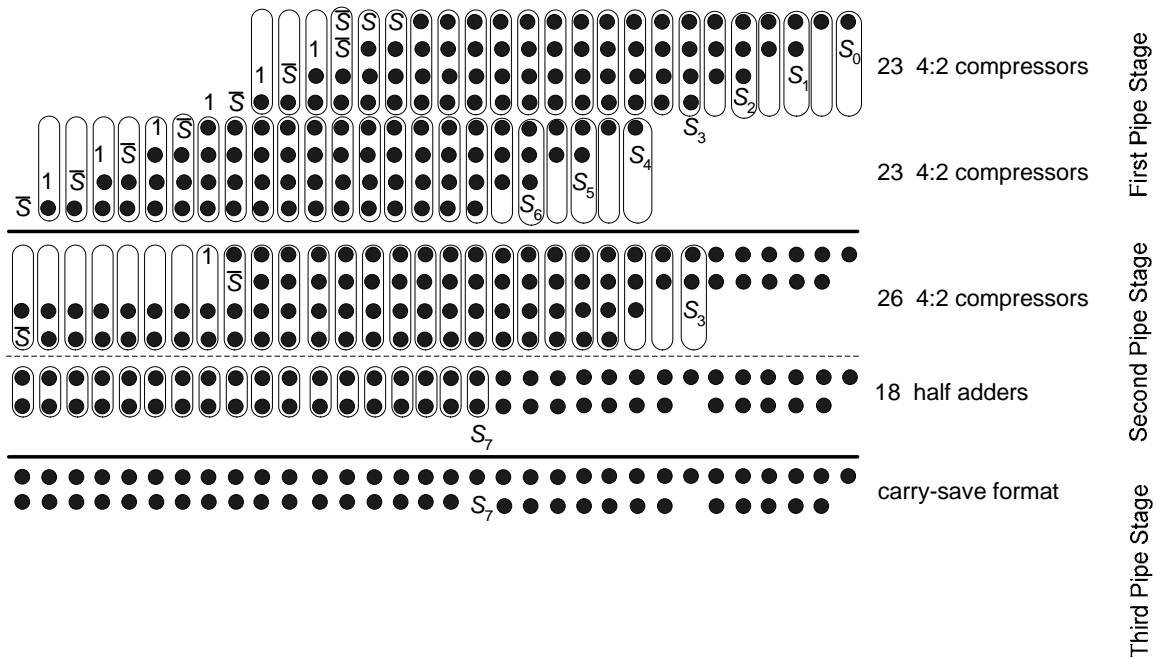


Figure 5.8: Partial Product tree reduction with 4:2 compressors and half adders: This figure shows the tree reduction of the  $16 \times 16$ -bit multiplication. Tree reduction is done in two pipe stages. The ovals around four dots are 4:2 compressors, and the ovals around two dots are half adders. The dark horizontal lines separate pipe stages. The dots between the first set of registers and the dotted horizontal line are the outputs of the first pipe stage. The dots between the dotted horizontal line and the second set of registers are the outputs of the row of 26 4:2 compressors and the 11 bits in the least significant position already in carry-save format. The inputs for the 4:2 compressors that do not cover any dots are tied to *Gnd*. The third pipe stage contains the two rows of bits to be used in a carry-propagate adder.

4:2 compressors are fed into a row of half adders to further reduce the height to two rows. The row of half adders (used as carry-save adders) reduces the height so the  $S_7$  bit will fit into carry-save format for the third pipe stage. There are a total of 62 registers at the end of the second pipe stage (35 registers for the bits from the half adders, 26 registers for the outputs of the 4:2 compressors that aren't fed into half adders, and 1 register for the  $S_7$  bit). Figure 5.8 shows the reduction of the partial product tree into carry-save format.

#### 4:2 compressor implementation

This section discusses the implementation of the 4:2 compressor used in the MAC unit and first discussed in Section 3.3.3. Oklobdzija showed an optimal interconnection between two full adder cells to build a 4:2 compressor, thus reducing the delay to three XOR gates in series [22]. The disadvantage of this implementation is the layout of the structure. If FAs are used as subcells, then

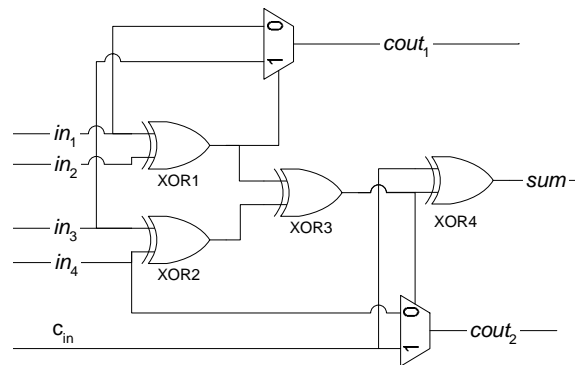


Figure 5.9: 4:2 compressor gate schematic

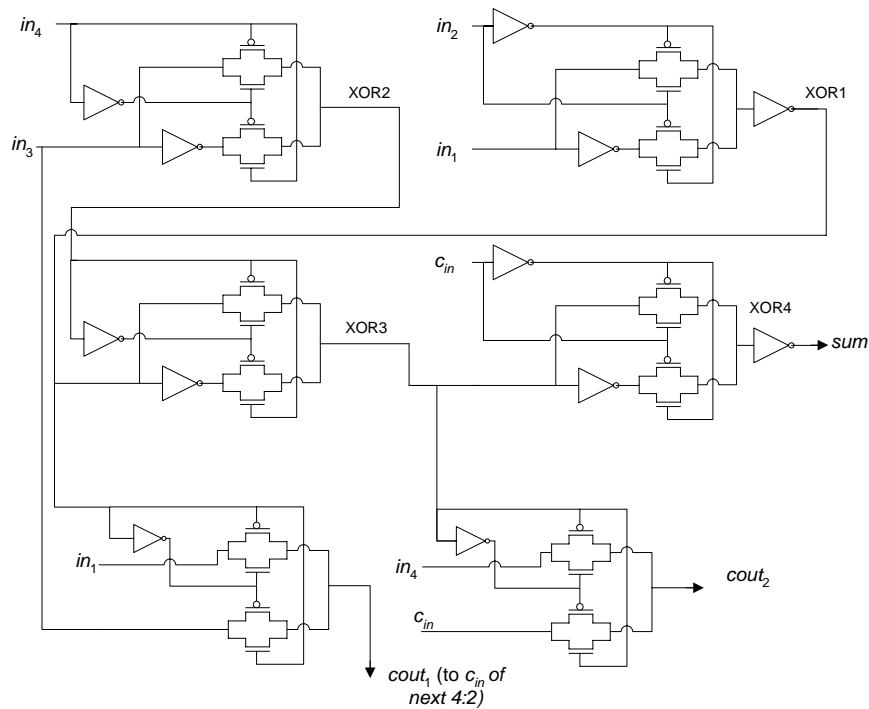


Figure 5.10: 4:2 compressor transistor implementation: This implementation of the 4:2 takes 48 transistors. The performance tests are run at 1.8 V and 40 °C with a 15 fF load at the three outputs. The inverters are shown as gates for simplicity (they are simply two transistors, one PMOS and one NMOS).

the shape and size of the 4:2 compressor is limited by the shape and size of two FAs. Since we are using full custom layout in our design, we could build a custom cell instead that implements the 4:2 compressor function and have the shape of the cell fit our layout. Another method to build a 4:2 compressor, presented by Gu and Chang [30], is built with four XOR gates and two 2-to-1 muxes. We implement this 4:2 compressor because its layout can be shaped to fit our design more easily than the implementation with two full adders. The Boolean equations for the outputs of Gu and Chang's 4:2 compressor are shown in Equations 5.6 – 5.8. The gate schematic is shown in Figure 5.9, and the transistor schematic is shown in Figure 5.10. We simulate the 4:2 performance at 1.8 V at 40°C and load the outputs with a 15 fF capacitance. Timing analysis for this gate shows that the latest arriving output signal is  $cout_2$  with a delay of 698 ps. This is from the transition for an input vector of  $in_4in_3in_2in_1c_{in}=11111$  to 01111. For this case,  $cout_2$  transitions from a 1 to a 0 when the input first changes because its mux is selecting the  $in_4$  signal. After two XOR delays the mux select signal changes and then chooses the  $c_{in}$ . After another mux delay,  $cout_2$  transitions back to a 1 from 0.

$$cout_1 = (in_1 \oplus in_2) \cdot in_3 + \overline{(in_1 \oplus in_2)} \cdot in_1 \quad (5.6)$$

$$sum = in_4 \oplus in_3 \oplus in_2 \oplus in_1 \oplus c_{in} \quad (5.7)$$

$$cout_2 = (in_4 \oplus in_3 \oplus in_2 \oplus in_1) \cdot c_{in} + \overline{(in_4 \oplus in_3 \oplus in_2 \oplus in_1)} \cdot in_4 \quad (5.8)$$

### Half adder implementation

Half adders (Section 2.1.1) are used in the second pipe stage of the MAC unit. This architecture proves to be very efficient because it is the minimum amount of hardware required to reduce the height of the tree, and none of its inputs are unused (tied to  $Gnd$ ). The gate schematic for the implemented half adder is shown in Figure 5.11. There is a slow input and a fast input for this cell, meaning that if we have unequal signal input arrivals, then the faster arriving input signal should be tied to the fast input and the slower arriving signal tied to the slow input. If an early input signal arrives and is tied to the fast input, the delay for the  $sum$  is 201 ps and for the  $cout$  it is 151 ps. If instead the earlier arriving signal is tied to the slow input, then the delay for the  $sum$  is 236 ps and for the  $cout$  it is 149 ps. The test conditions are the same as those for the 4:2 compressor. From the timing analysis of the 4:2 compressor, the  $cout_2$  signal should be tied to the slow input, and the  $sum$  to the fast input. Since this is not the critical pipe stage, the optimization of these connections is not so important, but in the higher level chip design there may be the use of “cycle stealing” from faster pipe stages. Cycle stealing is a method where pipe stages with less

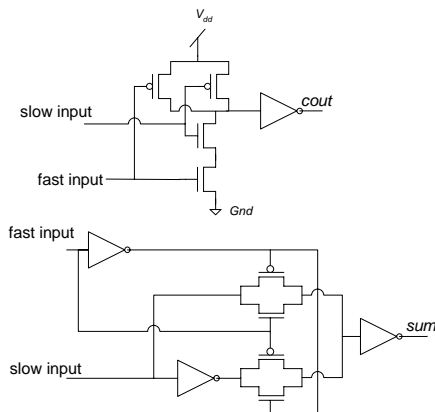


Figure 5.11: Half adder transistor implementation: The *cout* logic is a NAND followed by an inverter; the *sum* logic is an XOR with a driver. The fast input is tied to the controls for the transmission gates in the XOR so that they will be ready for the slow input; it is tied to the input farther away from the output node in the NAND so that only one node has to discharge when the slow input arrives.

logic (and therefore less propagation delay) have their clock pulses shortened so an adjacent pipe stage will have a longer clock pulse. This gives it more time to perform its logic. Since this second pipe stage is fast and the stage following this one will be slower, this stage is a candidate for cycle stealing. From the timing for the 4:2 compressor and the HA, we estimate the delay for the logic in the second pipe stage to be 936 ps.

### Testing the first two pipe stages

A simple way to verify the functionality of the first two pipe stages is to take the carry-save format from the output of the second stage and feed that into a 32-bit adder. It is easier to look at the output after it has been summed by the adder rather than the two quantities in carry-save format. We lay out a 32-bit ripple adder and look at its output, which is simply the result of a  $16 \times 16$ -bit multiplication operation. The ripple adder is easy to lay out (only one FA cell arrayed out 32 times). Also, note here that if an application requires just a  $16 \times 16$ -bit multiplier, one could follow the design of the Sections 5.2.1 and 5.2.2 and use a 32-bit carry propagate adder in the third pipe stage.

### 5.2.3 Final stage for the MAC unit

The third and final stage of the MAC unit completes the accumulation of the partial product tree. It contains a row of 40 AND gates, a row of 40 full adders, a 40-bit carry select adder, and

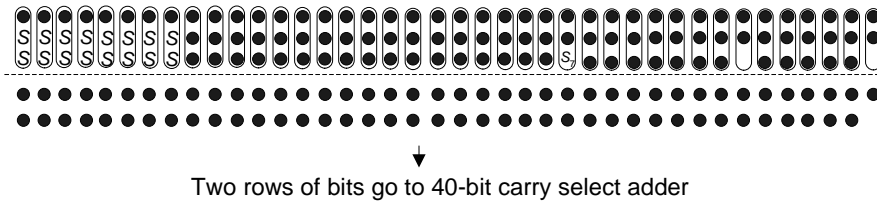


Figure 5.12: Dot diagram of bits in the third pipe stage: The ovals represent FAs used as carry-save adders. The 32-bit quantities from the second pipe stage (lower two rows of dots) are sign extended to make 40-bit quantities. The top row of dots are generated from the accumulator.

a 40-bit right shifter. During the execution of this final stage, the result of either a multiply/MAC instruction or a shift instruction is selected, thus the propagation delays for these instructions occur in parallel and the critical path is one of the two. This stage also contains 40 registers for the accumulator, and 16 registers for the multiplication result.

### AND gates and Full Adders

For the multiplication and MAC instructions, three rows of bits need to be accumulated (two rows from the second pipe stage and one from from the accumulator). The dot diagram for these three rows is in Figure 5.12. The accumulator result is only added for a MACL or MACH instruction. A row of 40 AND gates are used to select whether to use the accumulator result or not, and its output goes into the row of full adders. If *mac\_en* is low, then the top row of bits are all zeroes; if *mac\_en* is high, then the top row of bits are the contents of the accumulator. Full adders (first discussed in Section 2.1.2), are used to accumulate the three rows. They are used as carry-save adders (there is no ripple between the FAs) to accumulate the bits into carry-save format. Its three inputs are taken from the registers of the second pipe stage and the AND gate. The AND gate has a worst case measured propagation delay of 125 ps (at 1.8 V, 40°C, and a 15 fF output load). The full adder's implementation is shown in Figure 5.13. We can take advantage of unequal input signal arrivals with the full adder, for two of the inputs come straight from the pipe registers, and the other comes from the AND gate. If we hold the input signals to *A* and *B* of the full adder constant, the worst case delays for *sum* and *c<sub>out</sub>* are 180 ps and 172 ps, respectively (with the same test conditions as the AND gate). Otherwise, the worst case delay found for *sum* and *c<sub>out</sub>* are 385 ps and 189 ps, respectively (input vector transition of  $in_2in_1in_0 = 101$  to 100).

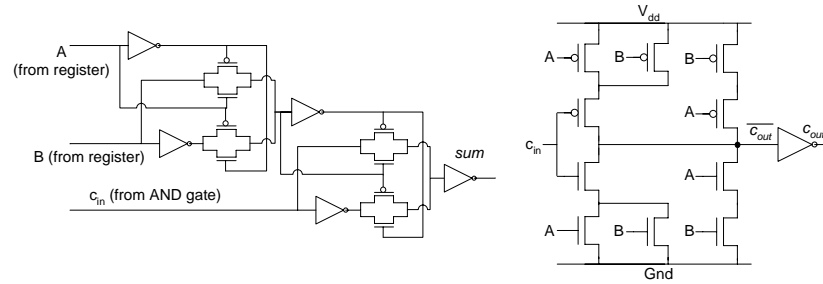


Figure 5.13: Full adder transistor implementation: The *sum* logic consists of two XOR gates in series, one without a driver and one with a driver. The *c<sub>out</sub>* logic consists of the mirror adder carryout logic discussed by Rabaey [4], with an inverter at the end (the mirror adder carryout logic generates  $\overline{c_{out}}$ ). The early arriving input signals should be fed into the inputs *A* and *B*, so there is at most one XOR delay when *c<sub>in</sub>* arrives.

#### 40-bit carry select adder

The outputs of the full adders are in carry-save format and are fed into the 40-bit adder for the final addition. Since two 40-bit quantities are a large amount to add, we implement the carry-propagate adder with a carry-select adder (first discussed in Section 2.2.4) to achieve a low propagation delay. An 8-bit CLA adder and a 16-bit CLA adder are used as subcells. The 8-bit CLA adder and the 16-bit CLA adder are used to make a 24-bit adder, with the *c<sub>out</sub>* of the 8-bit adder tied to the *c<sub>in</sub>* of the 16-bit adder. An 8-bit adder is already laid out for the address generators [31] in ASAP. The block diagram for the 24-bit adder is in Figure 5.14. The 40-bit carry select adder is built with one 16-bit CLA adder, two 24-bit adders, and a 24-bit 2-to-1 mux. There is redundant hardware in the 24-bit adder because the result of one of these adders will be unused in the final sum. The *c<sub>in</sub>* for the 24-bit adder is the *c<sub>out</sub>* of the 16-bit adder. However, instead of waiting for this *c<sub>out</sub>*, the *c<sub>in</sub>* for one 24-bit adder is tied to *Gnd* and the *c<sub>in</sub>* for the other is tied to *V<sub>dd</sub>*. This allows the addition for all three adders to occur in parallel. The *c<sub>out</sub>* of the 16-bit adder is now the select line for the mux. The block diagram for the carry select adder is in Figure 5.15. The 16-bit adder will complete its calculation before the 24-bit adder, however, there is some delay to assert the select line for the mux. The propagation delay of the 24-bit adder is still greater than the delay of the 16-bit adder and the mux, so the critical path will be through the 24-bit adder.

The 40-bit adder is laid out and extracted into an HSPICE netlist, but we analyze the performance of the 24-bit adder since it is the critical path. We find the mux select line is asserted in approximately 930 ps, so it will be ready before the outputs of the 24-bit adder are calculated. The input test vectors are *A*=0xFFFFFFFF and *B*=0x000001 with the least significant bit of *B* toggling.

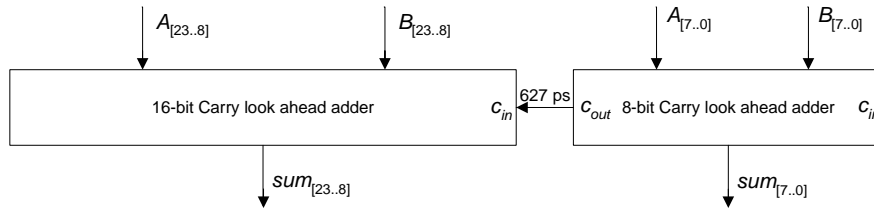


Figure 5.14: 24-bit carry propagate adder: The adder is built with an 8-bit carry look ahead adder and a 16-bit carry look ahead adder.

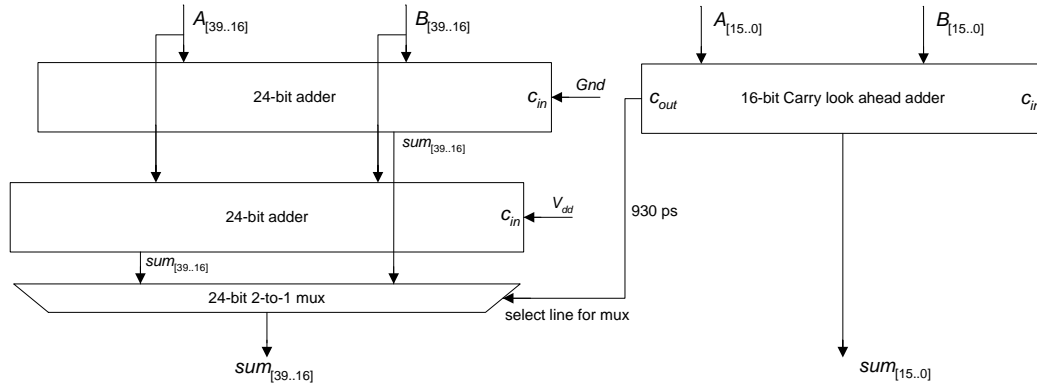


Figure 5.15: 40-bit carry select: The adder uses a 16-bit adder and two 24-bit adders.

The tests are run at 1.8 V, 40°C, and with a 50 fF load at the output of the mux. This large capacitance is meant to simulate the long wires the mux needs to drive. If the delay is too long, buffers can be added. The  $c_{out}$  for the 8-bit adder arrives in 627 ps, and the latest  $sum$  bit arrives in 1.150 ns. The additional delay for the 16-bit adder subcell in the 24-bit adder is only 541 ps (as opposed to the 854 ps critical delay reported in Section 2.4.6). We believe the reason for this is because while the 16-bit adder is waiting for its  $c_{in}$  signal, the  $A$  and  $B$  inputs have already propagated through the circuit and are settled. Thus when  $c_{in}$  arrives many of the nodes have already charged or discharged to their final value. The least significant 16 bits are calculated well before the most significant 24 bits, so the final value for the propagation delay of the 40-bit carry select adder is 1.250 ns. This 100 additional picoseconds is for the delay through the 40-bit mux and the charging of the inputs to the buffers after the mux.

#### 40-bit Accumulator right shifter

The 40-bit right shifter takes the result of the accumulator register as an input and produces a 40-bit shifted result. The purpose of this shifter is to be able to access all the bits in the accumulator. This is accomplished by reading the lower 16 bits of the accumulator, then right shifting the accumulator value to access its upper bits. Since AsAP uses 16-bit arithmetic, the shifter can shift up to 16 bits at a time. Using the barrel shifter architecture described in Section 4.2.3 for the ALU, this shifter requires five stages. This means the shifter can actually shift a maximum of 31 places, but 16 places is the most important and frequently used shift amount. For the ACCSHR (right shift accumulator) instruction, the shift amount is governed by the lower five bits of input *A*. For the multiply instructions, the accumulator value is preserved so the shift amount is zero. For the MAC and MACC instructions, the shift amount is a “don’t care” since the mux before the accumulator does not select the result of this shifter. The shifter is laid out and its functionality is verified in IRSIM. Since the input signals need to pass through five series transmission gates and these gates present a large resistance, the sizing for each transmission gate is  $20\lambda$  for each the PMOS and the NMOS, where  $\lambda=0.09\ \mu\text{m}$  in a  $0.18\ \mu\text{m}$  process. The inputs also need large buffers to drive the signals through to their outputs. When simulating the shifter we assume that the shift amount select lines are asserted to their final values, and take the 50% point of the post-buffered input signal to the 50% point of the latest output. The supply voltage is 1.8 V, the temperature is  $40^\circ\text{C}$ , and each output is loaded with a 15 fF capacitance. For a right shift of 16 places the propagation delay is 360 ps. Even if we take into account the delay for the buffers, this shifter delay is well under the delay for the 40-bit CPA, so it will not be in the critical path.

#### 5.2.4 Decode Logic

There are a total of four control signals for the MAC unit: MAC instruction enable (*mac\_en*), multiply high/low select (*mult\_hilo*), accumulator select (*acc\_sel*), and accumulator shift amount select (*shift\_sel*). Table 5.4 shows these signals and their values for the instructions that the MAC unit implements. The opcode for AsAP is six bits wide, and the upper two bits are used to specify a MAC/multiply/accumulator shift instruction, so there are four bits remaining for the four control signals. This means there is no decode logic required for the MAC unit, for one bit of the opcode is wired to each control signal.

| Instruction | Control signals |                  |                |                  |
|-------------|-----------------|------------------|----------------|------------------|
|             | <i>mac_en</i>   | <i>mult_hilo</i> | <i>acc_sel</i> | <i>shift_sel</i> |
| MULTL       | 0               | 0                | 0              | 0                |
| MULTH       | 0               | 1                | 0              | 0                |
| MACL        | 1               | 0                | 1              | X                |
| MACH        | 1               | 1                | 1              | X                |
| MACCL       | 0               | 0                | 1              | X                |
| MACCH       | 0               | 1                | 1              | X                |
| ACCSHR      | 0               | X                | 0              | 1                |

Table 5.4: Truth table for MAC control signals: *mac\_en* chooses whether or not to accumulate the product. It is asserted for a MACL and MACH instruction only. *mult\_hilo* selects between the upper or lower 16 bits of any MAC, MACC, or multiply instruction. *acc\_sel* selects either the 40-bit CPA result or the 40-bit shifted accumulator result. *shift\_sel* chooses the shift amount for the accumulator. For multiply instructions the accumulator is preserved, so the shift amount is zero. For ACCSHR instructions the shift amount is determined by the lower five bits of the input *A* (which are the lower five bits of the multiplicand in the other MAC unit instructions).

### 5.3 Layout and Performance

The MAC unit is laid out and occupies an area of 225,049  $\mu\text{m}^2$  with a height of 374  $\mu\text{m}$  by a width of 604  $\mu\text{m}$  in TSMC 0.18  $\mu\text{m}$  technology using scalable SCMOS design rules. This is the smallest rectangle that can fit around the whole unit, but there are large amounts of unused space that can be used for other circuits and routing. The estimated active area is 180,244  $\mu\text{m}^2$ . A total of five metal layers are used to lay out the MAC unit. The metal 5 layer is used for routing the signals from the first row of 4:2 compressors to the row of registers for the first pipe stage. It is also used to route the result of the accumulator to the row of 40 AND gates used in a MAC instruction. Thus, a total of 87 metal 5 tracks are used and unavailable for use in the global layout.

The layout is first extracted into an IRSIM netlist to test functionality. There are 20,524 transistors in the MAC unit (as reported by IRSIM). The first two pipe stages are tested using the method described in Section 5.2.2, where a 32-bit adder is used in the third pipe stage and 16-bit multiplication operations are run. Once the first two stages are verified, we perform functional tests on the whole MAC unit. The layout is then extracted into an HSPICE netlist for the performance tests. For all the performance tests the supply voltage is 1.8 V, the temperature is 40°C, and the transistor models are typical PMOS and typical NMOS for the TSMC 0.18  $\mu\text{m}$  process.

The MULTL and MULTH instructions are simulated with an input of  $A = 0x0000$  and  $B = 0x0000$  transitioning to  $A = 0x0001$  and  $B = 0xFFFF$ . This will cause the signals before the *mult\_out* register to go from 0x0000 to 0xFFFF. We probe the signals before the input to the register so we can view both the MULTL and MULTH result (only one of these results will be

| Instruction | Delay    |
|-------------|----------|
| MULTL       | 1.000 ns |
| MULTH       | 1.384 ns |
| MACL        | 1.636 ns |
| MACH        | 1.636 ns |
| MACCL       | 1.442 ns |
| MACCH       | 1.442 ns |
| ACCSHR      | 856 ps   |

Table 5.5: MAC unit performance: This table lists each instruction the MAC unit can execute and their delays. These delays are taken from the 50% point of the falling clock edge to the 50% point of the latest arriving output signal right before the register.

chosen, depending on the *mult\_hilo* control signal). The MULTL delay from the 50% of the falling clock edge to the latest arriving signal is 865 ps. For the MULTH it is 1.41 ns. The next set of inputs are  $A = 0x0001$  and  $B = 0x0000$ . This will cause the output before the register to go from  $0xFFFF$  to  $0x0000$ . The MULTL delay for this transition is 1.000 ns and for the MULTH instruction it is 1.384 ns. The energy consumed for a MULTL instruction is the same as for a MULTH instruction because both results are calculated at the same time but only one is selected. We measure the energy for the multiply instruction where the output transitions from  $0x0000$  to  $0xFFFF$ . The average current drawn is 18.3 mA over 2 ns, equaling a charge of 36.6 pC and 65.8 pJ of energy consumed. This results in a maximum power dissipation of 47.6 mW at the maximum operating frequency of 723 MHz at a 1.8 V supply voltage.

To simulate the MAC instructions, we begin by overwriting the accumulator with a MACC instruction. The value we initialize into the accumulator is  $0x0000000001$ , and this is accomplished with a MACCH instruction with input operands of  $A = 0x0001$  and  $B = 0x0001$ . The subsequent instructions are MACH instructions with input operands of  $A = 0x0001$  and  $B = 0xFFFF$ . This will result in a value of  $-1_{10}$  being added to the current accumulator of  $+1_{10}$ , and will cause the value written to the accumulator to  $0x0000000000$ . Maintaining the MACH instruction with the same operands for the next cycle, the value written to the accumulator will now be  $0xFFFFFFFF$  (the result of 0 minus 1). For the transition of the accumulator from  $0x0000000001 \rightarrow 0x0000000000$ , the MACH instruction has a delay of 1.636 ns (this is measured from the 50% of the falling clock edge to the latest arriving output signal). For the transition of the accumulator from  $0x0000000000 \rightarrow 0xFFFFFFFF$ , the MACH instruction has a delay of 1.558 ns. For both these cases the result for the *mult\_out* register has the same delay because the 16-bit bus for the high result is part of the 40-bit bus from the adder to the accumulator. We also measure the average current during each clock cy-

cle, and we found that the MACH instruction that causes the transition of the accumulator from  $0x0000000001 \rightarrow 0x0000000000$  draws the most current. For this instruction, the average current is 14.1 mA over 2 ns, equaling a charge of 28.2 pC and 50.8 pJ of energy consumed. This results in a maximum power dissipation of 31.0 mW at the maximum operating frequency of 611 MHz at a 1.8 V supply voltage.

The MACL instructions have the same propagation delay as the MACH instructions because for the same set of inputs both of these instructions write the same value to the accumulator, and their only difference is the result written to the *mult\_out* register. The propagation delay for the *mult\_out* register for a MACL instruction is shorter than for a MACH instruction because the lower sixteen bits of the adder are calculated sooner, however we cannot run the clock any faster because there is still a write to the accumulator that needs to take place.

To simulate the MACC instructions, we begin with a MACCL instruction with inputs of  $A = 0x0000$  and  $B = 0x0000$ , resulting in an accumulator value of  $0x0000000000$ . The subsequent instruction is a MACCL with inputs of  $A = 0xFFFF$  and  $B = 0x0001$ . This will result in an accumulator value of  $0xFFFFFFFF$ . The delay is 1.442 ns. This is followed by another MACCL instruction with inputs of  $A = 0x0000$  and  $B = 0xFFFF$ , which will overwrite the accumulator with  $0x0000000000$ . The delay is 1.286 ns. For this instruction, the average current drawn for the cycle where the accumulator transitions to  $0xFFFFFFFF$  is 14.3 mA over 2 ns, equaling a charge of 28.6 pC and 51.5 pJ of energy consumed. This results in a maximum power dissipation of 35.7 mW at the maximum operating frequency of 693 MHz at a 1.8 V supply voltage.

To simulate the ACCSHR instruction, we begin with a MACCL instruction that initializes the accumulator to  $0xFFFFFFFF$  ( $A = 0x0001$  and  $B = 0xFFFF$  accomplishes this). Then we follow it with an accumulator shift of 31 and probe the signals at the inputs to the accumulator. The latest signal arrives in 856 ps. The average current drawn for the cycle where a shift of 31 occurs is 15.1 mA over 2 ns, equaling a charge of 30.2 pC and 54.4 pJ of energy consumed. This results in a maximum power dissipation of 63.5 mW at the maximum operating frequency of 1.168 GHz at a 1.8 V supply voltage.

The slowest instructions in the MAC unit are the MAC instructions, and with a clock period of 1.636 ns the unit can be clocked at 611 MHz. Figure 5.16 shows the final layout for the MAC unit.

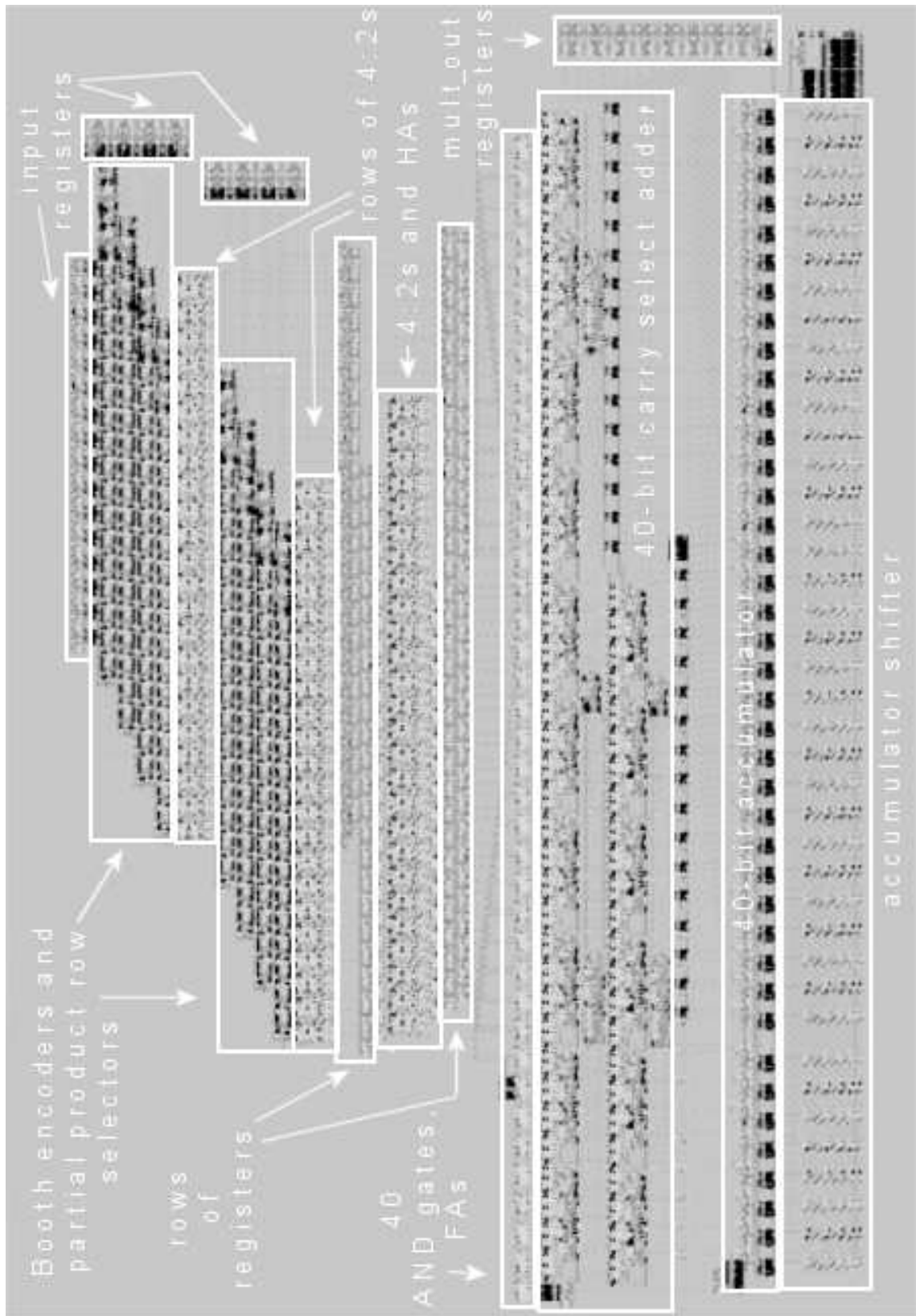


Figure 5.16: Final layout for the MAC unit

## 5.4 Summary

This chapter presents the MAC unit design and implementation for the AsAP chip. It utilizes Booth encoding, 4:2 compressors for partial product tree reduction, and a 40-bit carry select adder for final accumulation. The unit is pipelined into three stages and runs at a maximum clock frequency of 611 MHz.

## Chapter 6

# Conclusion

### 6.1 Summary

This thesis explores and analyzes fast adder algorithms and multiplication schemes, and utilizes them in the design and implementation of an ALU and a MAC unit. These two units perform nearly all the arithmetic operations in AsAP. The features of the ALU include the ability to compute logical operations, shifts, saturating additions and subtractions. The features of the MAC unit include Booth encoding, 4:2 compressors for partial product tree reduction, a 40-bit carry select adder, and a 40-bit accumulator shifter. The ALU runs at a maximum speed of 398 MHz and occupies  $56,936 \mu\text{m}^2$ , and the MAC unit is pipelined and runs at a maximum speed of 611 MHz and occupies  $225,049 \mu\text{m}^2$  in the TSMC  $0.18 \mu\text{m}$  process at 1.8 V and  $40^\circ\text{C}$ .

### 6.2 Future Work

The layout for both the ALU and MAC unit will be integrated into the global layout for AsAP, and the chip will be fabricated in the near future. It will be interesting to see how close our performance simulations match the actual speed of the chip. Also, the global layout for the processor is a huge task, and the area optimizations at this top level will present the greatest gains towards a chip with a low area. Many issues for this global layout will arise, such as:

- 1) Efficiently floorplanning and placing the blocks within each processor so that there is little wasted area,
- 2) Routing the global power, ground, and clock wires within each processor,
- 3) The maximum length of a global wire before repeaters need to be used, and

4) Making each processor “tileable” so that a 2-dimensional array of processors can be laid out with nearly no additional design time.

The ALU lies in the critical path of the processor, and there are certain optimizations that can be made to increase its speed. First, we can duplicate the zero-detect logic and place it before the saturating muxes in the adder block of the ALU. Thus, the saturate logic and zero-detect logic will run in parallel. The logic, word, and shift instructions are not in the critical path so it can still calculate whether its result is zero without increasing the ALU cycle time. We estimate this optimization will save between 200 and 250 ps. There is extra load on the wires before the saturating muxes, which will slow down the saturating adder, but after its output is calculated it no longer needs to go through the zero-detect logic. Another solution is to move the zero-detect logic into a subsequent pipe stage. This will save 250 ps, but will increase the branch shadow delay for the processor by one.

The MAC unit has a lot of unused area because there is no pitch matching between the second and third pipe stages. The width of the 40-bit carry select adder is what determines the width of the MAC unit. If in the global layout we cannot find a block that can utilize some of the unused area in the MAC, then the MAC unit is much bigger than it needs to be. Laying out an adder that has less width, and more importantly, the exact same width as the second pipe stage, will save about 25,000  $\mu\text{m}^2$  in area. However, we have to see if this “thinner” adder will be worth the time by first doing global floorplanning and determining if there are uses for the unused MAC unit area.

# Bibliography

- [1] J. Eyre and J. Bier, “DSPs court the consumer,” *IEEE Spectrum Magazine*, vol. 36, no. 3, pp. 47–53, Mar. 1999.
- [2] J. Eyre, “Digital signal processor derby,” *IEEE Spectrum Magazine*, vol. 38, no. 6, pp. 62–68, June 2001.
- [3] J. Eyre and J. Bier, “The evolution of DSP processors,” *IEEE Signal Processing Magazine*, vol. 17, no. 2, pp. 43–51, Mar. 2000.
- [4] J. M. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits, A Design Perspective*, Prentice Hall, Upper Saddle River, NJ, 2003.
- [5] B. M. Baas, “A parallel programmable energy-efficient architecture for computationally-intensive DSP systems,” in *Signals, Systems and Computers, 2003. Conference Record of the Thirty-Seventh Asilomar Conference on*, Nov. 2003.
- [6] Ryan W. Apperson, “A dual-clock FIFO for the reliable transfer of high-throughput data between unrelated clock domains,” M.S. thesis, University of California, Davis, Davis, CA, USA, Aug. 2004.
- [7] A. Chandrakasan, W. J. Bowhill, and F. Fox, *Design of High-Performance Microprocessor Circuits*, IEEE Press, Piscataway, NJ, 2001.
- [8] A. Weinberger and J. Smith, “A logic for high-speed addition,” in *National Bureau of Standards*, 1958, pp. 3–12.
- [9] I. Koren, *Computer Arithmetic Algorithms*, A.K Peters, Ltd., Natick, MA, 2002.
- [10] J. Sklanski, “Conditional-sum addition logic,” *IRE Transaction on Electronic Computers*, vol. EC-9, pp. 226–231, 1960.
- [11] O.J. Bedrij, “Carry-select adder,” *IRE Transaction on Electronic Computers*, June 1962.
- [12] “NC-Verilog,” [http://www.cadence.com/products/functional\\_ver/nc-verilog/index.aspx](http://www.cadence.com/products/functional_ver/nc-verilog/index.aspx).
- [13] “HSPICE,” <http://www.synopsys.com/products/mixedsignal/hspice/>.
- [14] “Magic – a VLSI layout system,” <http://vlsi.cornell.edu/magic/>.
- [15] “Digital integrated circuits – the IRSIM corner,” <http://bwrc.eecs.berkeley.edu/Classes/IcBook/IRSIM/>.
- [16] “Matlab,” <http://www.mathworks.com/products/matlab/>.
- [17] M.J. Schulte, P.I. Balzola, A. Akkas, and R.W. Brocato, “Integer multiplication with overflow detection or saturation,” *IEEE Transactions on Computers*, vol. 49, no. 7, pp. 681–691, July 2000.

- [18] M.R. Santoro and M.A. Horowitz, "SPIM: A pipelined 64x64-bit iterative multiplier," *IEEE Journal of Solid-State Circuits*, vol. 24, no. 2, pp. 487–493, Apr. 1989.
- [19] C.S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on Computers*, vol. 13, no. 2, pp. 14–17, Feb. 1964.
- [20] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 34, pp. 349–356, Mar. 1965.
- [21] A. Weinberger, "4:2 carry-save adder module," *IBM Technical Disclosure Bulletin*, vol. 23, Jan. 1981.
- [22] V.G. Oklobdzija, D. Vileger, and S.S. Liu, "A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach," *IEEE Transactions on Computers*, vol. 45, no. 3, pp. 294–305, Mar. 1966.
- [23] A.D. Booth, "A signed binary multiplication technique," *Quarterly Journal of Mechanics and Applied Mathematics*, vol. 2, pp. 236–240, 1951.
- [24] O.L. MacSorley, "High-speed arithmetic on binary computers," *IRE Transaction on Electronic Computers*, vol. 49, pp. 67–91, 1961.
- [25] G. W. Bewick, *Fast Multiplication: Algorithms and Implementation*, Ph.D. thesis, Stanford University, Stanford, CA, Feb. 1994.
- [26] J.L. Hennessy and D.A. Patterson, *Computer Architecture, A Quantitative Approach, Third Edition*, Morgan Kaufman Publishers, San Francisco, CA, 2003.
- [27] P. Bonatto and V.G. Oklobdzija, "Evaluation of booth's algorithm for implementation in parallel multipliers," *Signals, Systems and Computers*, vol. 1, pp. 608–610, Oct. 1995.
- [28] S. Shah, A.J. Al-Khalili, and D. Al-Khalali, "Comparison of 32-bit multipliers for various performance measures," *The 12th International Conference on Microelectronics*, pp. 75–80, Oct. 2000.
- [29] B.R. Zeydel, V.G. Oklobdzija, S. Mathew, R.K. Krishnamurthy, and S. Borkar, "A 90nm 1GHz 22mW 16x16-bit 2's complement multiplier for wireless baseband," *Symposium on VLSI Circuits, Digest of Technical Papers*, pp. 235–236, June 2003.
- [30] J. Gu and C-H. Chang, "Ultra low voltage, low power 4-2 compressor for high speed multiplications," *IEEE International Symposium on Circuits and Systems*, vol. 5, pp. 321–24, May 2003.
- [31] Omar Sattari, "Fast fourier transforms on a distributed digital signal processor," M.S. thesis, University of California, Davis, Davis, CA, USA, Aug. 2004.