Design and Layout of a Computational Field Programmable Architecture

by

Daniel I. Vranesic

A Thesis submitted in conformity with the requirements for the Degree of Master of Applied Science in the Department of Electrical and Computer Engineering University of Toronto

© Copyright by Daniel I. Vranesic 1998
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-34141-0
Design and Layout of a Computational Field Programmable Architecture

Daniel I. Vranesic
Master of Applied Science, 1998
Department of Electrical and Computer Engineering
University of Toronto

Abstract

Field Programmable Devices (FPDs) are a very popular medium for implementing digital circuits. They offer many positive features such as re-programmability, low design manufacturing cost and fast time to market.

Compute intensive applications used in signal and data processing are often implemented in FPDs. A primary reason for this is that FPDs are well suited to extracting the parallelism that often exists in these applications. FPDs are not ideally suited to this task, however, as they cannot efficiently realize common arithmetic structures such as adders, subtracers and multipliers.

We have designed and implemented a custom layout for a Field Programmable Architecture that is targeted to compute intensive applications. The architecture, called the CFA, is intended to offer efficient realizations of the arithmetic structures commonly found in compute intensive applications. Preliminary results from our detailed area model show that our architecture is on average 3-4 times more area efficient than conventional FPD architectures on a set of arithmetic circuits.
Acknowledgements

Many thanks to Professor Stephen D. Brown, for providing friendship, advice and support. And for missing the odd jump shot when I was guarding him.

Thanks also to Alireza and Guy for timely words of FPGA wisdom.

To the boys in the lab. In no particular alphabetical order...Alex, Derek, J. Greg, Kelvin, Mike, Rob, Robin and Steve. Squash, baseball, basketball, hockey, ridiculous debates, conversations and keg parties all conspired to keep me stay here longer than necessary, and to enjoy it more than I had any right to.

To my family. I'm done now.

To Kia. For love and understanding. I'm done now.


List of Figures

2.1 Four-input Look Up Table ........................................ 5
2.2 Xilinx XC4000 Series Logic Block ................................. 6
2.3 Altera FLEX 10K Series Logic Block ............................. 7
2.4 Segmented interconnect architecture ............................. 8
2.5 Altera FLEX 10K Series interconnect architecture ............ 9
2.6 Splash2 architecture ............................................. 10
2.7 PAM architecture .................................................. 11
2.8 TM2 board architecture .......................................... 13

3.1 General form of a PASM block .................................... 16
3.2 PASM bit-slice architecture ...................................... 17
3.3 Full adder with carry logic ...................................... 17
3.4 PASM bit-slice configured as an adder ......................... 18
3.5 PASM bit-slice configured as a subtracter ..................... 18
3.6 Array multiplier .................................................. 20
3.7 PASM configure for multiplication .............................. 21
3.8 The PASM block .................................................. 22

4.1 Simple logic gates used in a PASM .............................. 26
4.2 Alternate XOR gate ................................................. 27
4.3 SRAM cell schematic and layout based on two cross-coupled inverters 28
4.4 Two-input multiplexer designs .................................. 28
4.5 Four-input AND gate and bit-elect circuit layouts .......... 29
4.6 Various implementations of the carry logic circuit ...... 30
4.7 D flip-flop schematic ............................................ 31
4.8 D flip-flop layout ................................................ 31
4.9 Four-to-one multiplexer layout ................................ 32
4.10 Layout of PASM bit-slice ....................................... 32
4.11 Spice graph of PASM input and output signals ............. 34
4.12 Schematic of an output driver .................................. 35
4.13 Layout of an output driver ..................................... 35
4.14 Layout of a PASM block ....................................... 36
4.15 Schematic of a 16:1 multiplexer ............................... 37
4.16 Layout of a 16:1 multiplexer ................................ 38
4.17 Layout of a PASM block with routing multiplexers ....... 39
5.1 CFA architecture .............................................. 43
5.2 PASM cluster .................................................. 44
5.3 Cluster-level switch pattern ................................. 45
5.4 Cluster interconnect template .............................. 46
5.5 Dedicated wires to implement a shift register .......... 47
5.6 Dedicated wires between two PASMs in a multiply .... 48
5.7 Cluster-level interconnect. ................................ 49
5.8 PASM tile floorplan ......................................... 50
5.9 Metal levels in the PASM tile ............................... 51
5.10 Cluster-level switch physical locations ................. 52
5.11 Connection of multiplexer input switches to cluster-level wires .... 53
5.12 SRAM programming lines ................................. 54
5.13 Mapping of SRAM cells to data lines ...................... 55
5.14 Two bits of the 5-bit shift register ....................... 55
5.15 Mapping of SRAM cells to 10 control lines ............. 56
5.16 Four bits of the 10-bit shift register ..................... 57
5.17 Layout and floorplan of PASM cluster .................... 58
5.18 Wiring in a PASM cluster .................................. 59
5.19 H-Tree clock distribution network ........................ 59
5.20 Output waveforms of a single PASM in a cluster ....... 60

A.1 Map of cluster wires to multiplexer inputs ............... 65
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Gate area and average propagation delay.</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Area comparisons of various architectures on arithmetic benchmark circuits.</td>
<td>40</td>
</tr>
<tr>
<td>A.1</td>
<td>SRAM programming code for PASM</td>
<td>64</td>
</tr>
<tr>
<td>A.2</td>
<td>SRAM programming code for 4-input PASM multiplexer</td>
<td>65</td>
</tr>
<tr>
<td>A.3</td>
<td>Map of switches to cluster wires for left-column PASMs. * indicates that the multiplexer input connects to either the output or carry signal of the previous PASM in the cluster. Cluster wires labeled out that are not accompanied by * indicate outputs that come from the same PASM.</td>
<td>66</td>
</tr>
<tr>
<td>A.4</td>
<td>Map of switches to cluster wires for right-column PASMs. * indicates that the multiplexer input connects to either the output or carry signal of the previous PASM in the cluster. Cluster wires labeled out that are not accompanied by * indicate outputs that come from the same PASM.</td>
<td>67</td>
</tr>
<tr>
<td>A.5</td>
<td>PASM outputs connect to the cluster wires listed above. Refer to Figure 2.1 for map of PASMs relative to interconnect wires.</td>
<td>68</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Since their introduction in the 1980's, Field Programmable Devices (FPDs) have become an integral tool for digital circuit implementation. FPDs offer many advantages over conventional VLSI designs in areas such as manufacturing cost, time to market and in-system modification. However, the benefits of using FPDs come at a price. Re-programmable hardware is inferior in both speed and area compared to full custom VLSI circuit implementations.

FPDs implement circuits by connecting a number of generic logic elements with a flexible interconnection network. The network consists of a series of wires joined through programmable switches, whose capacitance and resistance contribute to the relative slowness of FPD-implemented circuits. The interconnect of an FPD, which can account for close to 70 percent of its total area, is also largely responsible for the fact that FPDs are larger than custom VLSI circuits.

Traditional FPDs can be divided into two classes: Field Programmable Gate Arrays (FPGAs) and Complex Programmable Logic Devices (CPLDs). Both types of devices contain logic blocks, connected by a programmable interconnect. FPGA logic blocks typically rely on Look-Up Tables (LUTs) or multiplexers to provide functionality, while CPLDs use AND-OR planes.

FPDs are used for a wide range of applications. [Sit] shows that 37 percent of all
FPDs are used in communications, 31 percent are used in data processing, 15 percent for industrial/instrumentation applications, 12 percent in networks and the remaining 5 percent are used in other applications. Clearly, FPDs are versatile enough to be used in many diverse areas.

One area in which FPDs are increasingly being used is the area of computation. Many applications such as image processing, which involve performing a series of arithmetic operations on a set of data, are very well suited to FPDs. The reason is that FPDs can be configured to take advantage of the parallelism that is present in this type of application. Several FPD-based compute engines have implemented computational applications. The speed performance of these applications has exceeded that of the same applications running on the fastest workstations, despite running at clock speeds that are much slower in comparison.

While these FPGA-based compute engines were successful from a speed perspective, they suffered from various shortcomings. Many machines were built using commercial FPD devices that contain generic logic elements. As a consequence, the area of a design was often significantly larger than a comparable design implemented in custom VLSI. Put simply, the implementations were fast, but large.

We believe that an approach to solving the area inefficiency of FPGA logic blocks at building certain arithmetic structures is to rethink the contents of a logic block. This thesis discusses the layout of a new architecture, called the Computational Field Programmable Architecture (CFA), that is aimed at overcoming this problem.

1.2 Objectives

The objective of this thesis is to evaluate the chip area requirements of the CFA, by creating a VLSI layout of part of a CFA chip. Using a 3-layer metal 0.5 \( \mu \text{m} \) CMOS process, a layout of the main logic block in the CFA has been created. To obtain an estimate of the area needed for routing resources, an initial routing architecture for interconnecting the logic blocks is presented and used in the chip layout.
1.3 Overview

This thesis is organized as follows. Chapter 2 contains background information on FPD-based computing. Chapter 3 discusses the architecture of the CFA. Chapter 4 gives details of the layout of the CFA logic block, called a Partial Adder/Subtracter/Multiplier (PASM). Chapter 5 presents the design and layout of the lowest level of the CFA interconnect. Chapter 6 offers conclusions and suggestions for future work.
Chapter 2

Background

Since their advent in the early 1980's, field programmable devices (FPDs) have spawned several important areas of research. One of the more interesting areas is that of FPGA-based computing. The idea is that a processing unit consisting of FPGAs can be mated to a conventional processor under the assumption that performance for certain types of applications can be sped up compared to that on a high-speed workstation. Many diverse FPD compute engines have been designed, each with its own unique features that have contributed to the knowledge in this new and exciting research area. This chapter discusses previous work in the field of FPGA-based computing and terminology related to the field.

2.1 FPGA Terminology

An FPGA can loosely be described as a set of programmable logic blocks connected by programmable interconnection resources. The nature of the logic blocks can vary greatly depending on the device, as can the architecture of the interconnect. The following two sections will describe terminology and various commercial architectures of both logic blocks and interconnection networks.
2.1.1 Logic Block Architecture

The logic block in an FPGA is responsible for providing the functionality of a digital circuit. Many commercial FPGAs use look-up tables as the basis of their logic blocks. A look-up table (LUT) comprises memory cells that store the output values of the truth table for a given function. Figure 2.1 [Bro95] shows an example of a 4-input LUT. The inputs $a$, $b$, $c$ and $d$, index a tree of multiplexers which send the correct function value to the output pin, $f$.

For a function with $k$ variables, $2^k$ memory cells are needed to represent the truth table. [RFLC90] showed that in terms of area efficiency, the best number of inputs to a look up table is about 4. This is the number of LUT inputs used in many commercial FPGAs.

FPGA logic blocks must also provide a way of implementing sequential circuits. This is accomplished in many devices by including a flip-flop at the output of the logic block. [RFLC90] showed that including a D type flip-flop in a logic block is superior in terms of area compared to building a similar structure out of logic blocks.

An example of an FPGA that is popular for use in industry is the Xilinx XC4000 [Xil96] series. The logic block, shown in Figure 2.2, contains 3 LUTs and 2 flip-flops. Some functions of as many as 9 inputs may be implemented, as well as any function...
of 4 or 5 inputs, or any two functions of 4 inputs. The LUTs may also be programmed to act as conventional memory, thereby providing support for data storage on chip.

The Altera FLEX 10K [Alt96] series FPGAs are also widely used. The FLEX 10K logic block architecture, shown in Figure 2.3, contains a single 4-input LUT. Both the XC4000 and FLEX 10K logic blocks contain dedicated circuitry for use with arithmetic building blocks, such as adders.

2.1.2 Interconnect

The design of the interconnection resources in an FPGA is important, because it can make up anywhere from 50-90% of an FPGA’s total chip area. As well, circuit performance directly relates to the interconnect architecture, because wire delay and programmable switch capacitance can be significant. [Fra92] shows that interconnect delays can account for up to 60% of the signal propagation time in FPGAs. Routing architectures can vary as widely as logic block architectures. This section describes
two common routing architectures found in commercial devices today.

**Segmented Architectures**

The Xilinx XC4000 series FPGAs use an array model for their logic block arrangement. This scheme is depicted in Figure 2.4. The logic blocks, labeled $L$, are placed in an array, separated by horizontal and vertical routing channels.

Each routing channel consists of $W$ wires. Different routing wires connect through a *switch* block, shown in part(a) of Figure 2.4. The switch block in the figure is of degree 3, meaning that any wire entering the switch block can connect to any of three other wires. Switch blocks are located at intersections of horizontal and vertical routing channels.

Access to a logic block is through a set of programmable switches that connect the logic block inputs to the channel segment wires. The switches reside in a *connection* block, shown in part(b) of Figure 2.4. The switches may take different forms; one method is to implement them as pass transistors controlled by an SRAM cell. Another method is to use a multiplexer, with the multiplexer inputs connected to the interconnect wires, the multiplexer output connected to the logic block input pin, and the multiplexer select lines controlled by SRAM cells. The connection block of
Figure 2.4 is of degree 2, meaning that each input to the logic block can connect to 2 interconnect wires.

The flexibility of this type of interconnect is dictated by three factors: the number of wires per channel, the degree of the connection block and the degree of the switch block. Altering these parameters affects how easy it is to successfully implement a circuit within the constraints of the chip architecture.

Hierarchical Architecture

An example of a hierarchical architecture is the Altera FLEX 10K FPGA. Eight logic elements, their carry and cascade chains, control signals and the local interconnect combine to form what is known as a Logic Array Block (LAB), shown in Figure 2.5.

Logic elements within the same LAB can be interconnected using the local interconnect. Logic elements in different LABs have to be connected using the next
higher-level interconnect, which spans the entire chip. Logic elements drive both the Local Interconnect and the FastTrack (chip-wide) Interconnect.

### 2.2 FPGA-based Computing

Parallel computing is a concept that has been around for a long time. Many solutions exist to take advantage of application parallelism such as multiprocessors, vector processors, systolic array processors and others. These types of processors have all proven to be effective at exploiting parallelism. However, they force an application to adapt to their architecture, which can potentially limit performance. Consider the case of an 8-way vector processor, which is a fixed architecture capable of performing 8 parallel instructions. If this type of machine is used to run an application that has a larger degree of parallelism than the machine can exploit, potential speed performance is lost.

Having a machine that is made out of reconfigurable logic offers a solution to this...
problem since the hardware can be tailored to the application that it is built for. The full parallelism of the application can be exploited by the hardware, because the hardware is not fixed in any way.

The next section will discuss various FPGA-based compute engines that have been built with the goal of exploiting data-parallel applications.

**SPLASH**

The SPLASH FPGA-based compute engine, developed at the Supercomputing Research Centre in Maryland, USA, was a systolic processing system that could also act as a SIMD processor. Each board of the system contained 32 Xilinx 3090 FPGAs connected in a linear array. The array could also be configured as a ring by connecting the array edges together. The board was connected via two FIFOs to a Sun Sparc workstation.

Although SPLASH was successful as a systolic processor, the designers noticed that it should be capable of more, but was held back by various architectural shortcomings. These shortcomings were addressed in the successor version to SPLASH, called SPLASH2, shown in Figure 2.6. SPLASH2 [JMAD92], built in 1992, upgraded the host interface, which had been a serious bottleneck in SPLASH,
to an SBus connection which increased the bandwidth between the SPLASH2 board and the host by a factor of 8. The chips used on the boards were also upgraded to the (then) new Xilinx 4010. The architecture of the boards was also changed somewhat. Memories were expanded to have one for each FPGA, the connection between FPGAs became more flexible and the number of FPGAs on a single board decreased from 32 to 17. The number of boards in a single system was scalable to 256. Highly parallel applications such as genetic database searching and real-time video processing have been sped up by several orders of magnitude compared to scalar processor execution times, despite running at relatively slow clock speeds.

PAM

Another project continuing the work on FPGA-based compute engines was the Programmable Active Memory (PAM) project [VBR+96] at the DEC Paris labs in 1992. The PAM concept is to connect a co-processor, which looks like a high-speed memory

![PAM Architecture Diagram](image)

Figure 2.7: PAM architecture.
module, to the host machine. The co-processor, shown in Figure 2.7, consists of an array of 16 processing elements (M), 4 banks of 1 MB blocks of SRAM, as well as glue logic. Everything on the board is implemented in Xilinx 3090 FPGAs. Dynamic reconfiguration is a major feature of the PAM project and because the board appears to the host as a simple memory module, programming is accomplished by including the processing configuration in the data stream generated by the host processor.

The DECPeRLe-1, as the first incarnation of PAM is called, is quite successful at speeding up certain applications. RSA cryptography runs at an order of magnitude faster than any existing implementation. Interestingly, a particle physics application called a Hadron Calorimeter has been implemented in a PAM and is the only one to meet strict performance limits established by the European Laboratory for Particle Physics (CERN).

**Teramac**

Teramac [ACC+95], designed at Hewlett Packard in 1995, is similar to both the SPLASH2 and PAM projects in that it consists of many FPGAs which can efficiently implement parallel algorithms and achieve high levels of speed-up. It is different in one aspect, however, which is size. A full Teramac system provides a programmer with upwards of 1 million usable gates. As a consequence, not only is Teramac useful as a large configurable compute engine, it is also capable of large-scale hardware emulation.

The Teramac is built using a custom FPGA called PLASMA. The PLASMA architecture has very generous routing resources that enable Teramac to implement almost any circuit topology. Teramac boards also contain four dual-ported two mega-bit by 32-word RAMs [CAC+96].

A full scale Teramac machine consists of a number of boards, each one containing 27 custom FPGAs capable of providing 64K gates each. The maximum clock speed of the Teramac machine is 1 MHz. However, given that thousands of operations can be executed in a single clock cycle, the low clocking speed becomes less of a factor in system performance.
Another feature of the Teramac is its efficient, seamless compiler. Designs can be translated from a high-level design (VHDL) to a placed and routed circuit in under one hour, which is impressive considering the potential size of circuit designs. Although the Teramac is an important step towards the development of co-processors, few have been built because of the high monetary cost.

Transmogrifier 2

More recently, other large scale co-processors/hardware emulators have been developed. The Transmogrifier 2 is a large field-programmable system with a logic capacity in excess of 2 million gates, implemented using Altera 10K50s [LGvI+97]. The TM2 has a very flexible interconnect, a large amount of RAM and the ability to run circuits at close to 10 MHz. The general board architecture is shown in Figure 2.8. It features two Altera FPGAs and four I3 interconnection crossbars. A backplane is used to connect the boards in the TM2, with communication at 25 MHz.

Programming of the TM2 can be accomplished in a matter of seconds. The
software developed for the TM2 is designed to offer a seamless package that transforms high level (VHDL) user designs to a machine configuration. Plans call for a full system of 16 boards, at 2 FPGAs per board.

CFA
This chapter discussed several machines that were effectively used to implement arithmetic algorithms. We believe that an architecture that is specifically intended to efficiently realize common arithmetic operations such as addition, subtraction and multiplication will have substantial area savings compared to general FPGA-based compute engines. Our proposed architecture, called the Computational Field Programmable Architecture (CFA) is designed with this goal in mind.

The next chapter discusses the details of the CFA architecture including the logic elements and the interconnection network.
Chapter 3

PASM Architecture

Conventional FPGAs have been used to exploit parallelism of computational applications such as image and digital signal processing. We believe that by altering the basic structure of an FPGA, we can obtain better speed performance and density than that achieved by conventional FPGA architectures. The proposed architecture is called Computational Field Programmable Architecture (CFA).

The CFA, whose original design appears in [Kav98], comprises two main components, called the LUTB and PASM blocks. LUTB stands for look-up table block, and is a set of conventional 4-input look-up tables that are grouped together. The PASM block, or partial adder/subtractor/multiplier is more specialized hardware and is the focus of this thesis. It offers a platform on which common arithmetic structures such as adders, substracters, multipliers and comparators can be efficiently implemented.

The basic function of the PASM block is as follows. The inputs are two 4-bit vectors. The output is also a 4-bit vector. The inputs can be added or subtracted. They may also be used in the bitwise logical operations AND, NOR, XOR and XNOR. An additional 1-bit input is used when the PASM is configured as a multiplier. This chapter gives a detailed description of the PASM architecture.
Figure 3.1 gives a general indication of the structure of a PASM block. It consists of 4 bit-slices stacked vertically, labeled in the figure as BS0, BS1, BS2 and BS3. Since we wish to make adders, subtracters and multipliers out of PASM blocks, we need carry information as well. Hence, there is a carry-in and a carry-out bit. The data input labeled \( m \) in the figure is used for multiplication.

Each PASM bit-slice is capable of performing addition, subtraction, and the bitwise AND, NOR, XOR and XNOR functions on the bit-slice inputs. The logic gates in a bit-slice are shown in Figure 3.2. The bit-slice can assume different configurations as determined by a set of SRAM cells, shown as red rectangles in Figure 3.2. The edge-triggered D flip-flop (DFF) enables synchronization and pipelining. The 4:1 multiplexer is used to save output pins by multiplexing 4 distinct logic outputs onto a single pin.

The bit-slice data inputs are labeled \( x_i \) and \( y_i \). The multiply select bit, used when the PASM is configured for multiplication, is labeled \( m \). The carry-in bit, labeled \( c_i \) and the carry-out bit, \( c_{i+1} \), are also shown. The output data bit is labeled \( \text{out}_i \).
3.1.1 Addition

A 1-bit full adder can be implemented by the circuit shown in Figure 3.3. Figure 3.4 shows how this circuit is realized in a PASM bit-slice. The active gates and wires are shown in cyan, and the gates that are not needed for the adder are shown in black.

The 2-input multiplexer that is controlled by the SRAM cell labeled AS/M (Add Subtract/Multiply) selects either the $m$ input or a logic 1. The $m$ input is used for multiplication, as will be explained in Section 3.1.3. For the adder, this multiplexer always propagates a logic 1. The rightmost XOR gate, which generates the sum bit of the adder, is available as the output of the bit-slice, if selected through the 4:1 multiplexer. The output can be selected as combinational or registered, depending on the value stored in the output select SRAM cell, labeled OS in the figure. Observe that the carry-input, $c_i$, is provided to the PASM bit-slice in its complemented form. This is done because of the way that the carry-logic is implemented, which will be discussed in detail in Section 3.2.1.
3.1.2 Subtraction

A subtracter is very similar in function to a full adder, assuming that one uses the 2's complement number representation. The conversion of a number from unsigned format to 2's complement format is straightforward. It is done by inverting all the bits then adding 1. In a PASM, this is accomplished by using an inverter for each input bit, and forcing a 1 on the carry input of the least significant bit. Although both the $x$ and $y$ inputs are available in true and complemented forms, only one of $x$ or $y$ should be inverted.

Figure 3.5 shows a PASM bit-slice with the active gates and wires needed for subtraction highlighted in cyan. The only differences between this configuration and that of the adder is that the complement of $y$ is now selected by its 2-input multiplexer and the carry-in bit of the least-significant bit-slice is forced active (0 in
the figure, since carry-in is inverted). We are assuming that the bit-slice shown is the least-significant bit-slice.

3.1.3 Multiplication

Multiplication is more expensive from a logic perspective than addition and subtraction. An example of binary multiplication on which the PASM multiplier is based is shown in Figure 3.6(a) [CHZ97]. The multiplicand \( X \) is multiplied by each bit of the multiplier \( M \), to form 4 summands. If the multiplier bit is a 1, the multiplicand is written as a summand in the appropriate location. If the multiplier bit is a 0, a row of zeroes is written. The addition of the four summands forms the product. The product is at most \( 2 \times n \) bits wide for the multiplication of two \( n \) digit numbers.

Figure 3.6(b) gives an example of how this type of multiplier can be realized in hardware as an array. The principle is that each row of the array generates its summand, and adds the summand to the incoming partial product. The incoming partial product to that row represents the sum of all the previous summands. Generating the correct summand bit is accomplished by ANDing the bit of the multiplier and the bit of the multiplicand. A full adder sums the summand bit to the bit of the incoming partial product. The logic cell at the bottom of part(b) shows how to combine these operations in the array multiplier.

Figure 3.7 illustrates how such a logic cell can be implemented in a PASM bit-slice. Again, the functioning wires and gates are shown in cyan. Notice that now, the \( m \) input is ANDed with the \( x \) input. The rest of the circuit functions as a full adder, with \( y \) selected in its uncomplemented state. Thus, the PASM bit-slice implements a single cell from Figure 3.6(b). Note that to build a \( 4 \times 4 \) multiplier as shown in Figure 3.6(b) requires 4 PASMs, because each PASM contains 4 bit-slices.

3.1.4 Bit-wise Boolean Operations

Several bit-wise boolean operations are available in a PASM bit-slice, including the AND, NOR, XOR and XNOR operations. The AND operation is simply \( g_i \) in Figure
(13) Multiplicand X
1 1 0 1
1 0 1 1
---
(11) Multiplier M
1 1 0 1
1 1 0 1
0 0 0 0
1 1 0 1
---
(143) Product P
1 0 0 0 1 1 1 1

(a) Manual multiplication algorithm

(b) Array implementation

Figure 3.6: Array multiplier.
3.7, while the XOR operation is $p_i$. The NOR operation can be computed as $\bar{y}_i \cdot \bar{x}_i$, while the XNOR function is $\bar{x}_i \oplus y_i$ or $\bar{y}_i \oplus x_i$. These functions, along with the sum output are inputs to the 4:1 multiplexer. A fourth input, representing either the equal (EQ), less than (LT) or greater than (GT) functions acting on the two four-bit data vectors is also available. Figure 3.7 shows this fourth input as the GT signal, which appears in the second bit-slice. The LT and EQ signals are present in the third and fourth bit-slices, while the top bit-slice has its fourth multiplexer input tied to ground. The input selection of the 4:1 multiplexer is controlled by the two SRAM cells labeled SEL0 and SEL1. The details of how the EQ, LT and GT signals are generated are discussed in Section 3.2.2.

### 3.2 PASM Architecture

To this point, only a single bit-slice of the PASM architecture has been examined. This section deals with the features of the PASM architecture as a whole, including the reconfiguration SRAM cells, the carry chain and the 4-bit comparator logic.

A single PASM block consists of four bit-slices, as illustrated in Figure 3.8. Logic to generate the EQ, LT and GT signals is located at the bottom. Programming is done by 9 SRAM cells, which are represented by red squares.
Figure 3.8: The PASM block.
3.2.1 Carry Logic

The carry logic in a PASM is implemented with the intention of making propagation as fast as possible and implemented with as little logic as possible. The carry-out signal is derived by rippling the carry through the four PASM bit-slices. The delay for this case is not severe, however. Since the propagate and generate signals are all derived in parallel (as soon as the input lines are driven), the delay is equal to four carry-logic unit delays plus the delay to generate the $p_i$ and $g_i$ signals. A carry-skip gate (four-input AND gate at the bottom of Figure 3.8) is used to determine if a propagate condition exists for all pairs of data inputs. If so, the carry-in signal is immediately propagated as the carry-out signal of the PASM.

3.2.2 Comparator Logic

The PASM block can function as a 4-bit comparator. The available operations are $x = y$ (EQ), $x < y$ (LT), $x > y$ (GT), $x \neq y$ (NEQ), $x \leq y$ (LTEQ) and $x \geq y$ (GTEQ). The logic for these operations is located at the bottom of the PASM block of Figure 3.8. To minimize the number of output pins, the comparator outputs are multiplexed on the output pins of three of the PASM bit-slices. Consequently, three comparison outputs are available at a time. Either the EQ, LT and GT operations, or the NEQ, LTEQ and GTEQ operations. Note that the NEQ, LTEQ and GTEQ outputs are generated by inverting the EQ, GT and LT outputs, respectively. This is accomplished using XOR gates with the output of the SRAM labeled \textit{INVC} (Invert Comparator) acting as the toggle bit.

If $x$ and $y$ are equal and the PASM is configured as a subtracter, then $p_i$ is a logic 1. The four-input AND gate then indicates whether or not all four input bit pairs are equal. Consequently, the EQ signal is taken from the output of the four-input AND gate. If the result of $x - y$ is positive, meaning that $x > y$, the carry-out will be a logic 0. Conversely, if the result is negative, the carry-out will be a logic 1. Hence, the GT output can be generated using the carry-out signal. Lastly, the LT operation can be represented as $LT = \overline{EQ} \cdot GT$, which is the NOR of the EQ and GT signals.
3.2.3 Programming Bits

The PASM is intended to be a flexible logic block capable of performing a number of different operations. SRAM cells were chosen as a means of offering easy reprogrammability. Only 9 are needed to program a PASM.

One SRAM cell is dedicated to selecting either the combinational output signal or the registered version; in Figure 3.8 it is labeled OS (Output Select). Another two SRAM cells control which signal each 4:1 multiplexer selects, and they are labeled SEL0 and SEL1. An SRAM cell is also needed to determine whether the EQ, GT, and LT outputs, or the NEQ, LTEQ and GTEQ outputs are selected. It is labeled INVC. Recall that for a subtract operation, a logic 0 must be forced on the carry-in bit. This is accomplished by another SRAM cell labeled S/A (subtract/add). However, for addition, the true input carry is needed, so the output of the S/A SRAM cell and the true carry input signal are connected to a multiplexer that is controlled by the SRAM cell labeled Cntl (carry control). For multiplication, it is necessary to AND the multiplicand bit \( m \) with the \( x \) input, however for addition/subtraction, the \( x \) input should not be affected by the value of \( m \). Therefore the \( m \) bit and a logic 1 are connected to a multiplexer controlled by the SRAM cell labeled AS/M (add subtract/multiply). Lastly, the \( x \) and \( y \) inputs need to switch between their true and complemented forms. This is done by connecting \( x \) and \( \overline{x} \) to a multiplexer controlled by the SRAM cell labeled XSEL in Figure 3.8. The same is true for \( y \) and \( \overline{y} \). Their multiplexer is controlled by the output of a second multiplexer. This second multiplexer has \( m \) and the output of the SRAM cell labeled S/A (subtract/add) as its inputs and is controlled by the SRAM cell labeled addsub. The reason for using this second control multiplexer is to allow \( m \) to act as a toggle bit controlling the mode of the PASM. If \( m \) is a logic 1, the PASM will behave as a subtracter, while a logic 0 will make the PASM act as an adder. For this to work, the carry-in bit must be the inverse of \( m \). This can be accomplished using LUTBs external to the PASM.
Chapter 4

PASM Tile Layout

This chapter discusses the layout implementation of a PASM tile. The layout of individual gates is discussed first, followed by the layout of a PASM bit-slice. The PASM block layout and routing multiplexers are then discussed. Area comparisons to existing commercial architectures are presented in the final section.

4.1 Methodology

The CAD tool used for this project is the BALLISTIC analog layout language. BALLISTIC [ODJ+96] is a compiler that generates Mentor Lx code running in the GDT layout editor. It is a hierarchical language that permits layouts to be constructed in a generic form that is not technology dependent. Porting of circuits that have been generated using BALLISTIC between different technologies can be accomplished with relatively little effort, because objects are placed relative to each other as a function of generic design rules.

BALLISTIC takes the form of a programming language, including the use of loops, variables, arrays, arithmetic statements and conditional structures. These features permit a high degree of versatility in the code structure and, given that circuit layouts are often repetitive, makes circuit designing relatively efficient.

Objects in BALLISTIC are placed using relative locations, although absolute co-ordinate location is also supported. When an object is placed, it is done so in
relation to another previously placed object, at a specified distance. Therefore, when adjustment is necessary or an object must be moved, relative spacing between affected objects is preserved. This feature makes it easy to either adjust a layout, or change the technology in which it is synthesized. This feature is useful for building the CFA layout since it is still in its early stages of development. We have been able to create a functionally verified layout which can be easily ported to different technologies, or altered with little difficulty.

The CFA layout is implemented using a 0.5 µm, 3-layer metal CMOS technology provided by the Hewlett-Packard Company.

### 4.2 PASM layout

The PASM layout was done in a hierarchical fashion, building from basic gates. As mentioned in Section 3.1, the PASM consists of 4 bit-slices. The layout of a bit slice is discussed below.

#### 4.2.1 Bit-Slice Components

Since most of the circuit elements in the PASM bit-slice are small (pull-up and pull-down networks consist of two or fewer transistors), little effort was put into sizing the transistors to optimize circuit speed. Our primary objective was to create a layout that is as compact as possible. Consequently, close to minimum-sized transistor gate widths are used in both NMOS and PMOS transistors. All gate widths are two times the minimum allowable gate width for the 0.5 µm process, which is 0.6 microns. The

![Figure 4.1: Simple logic gates.](image)

(a) two-input AND gate  
(b) buffer  
(c) NOR gate
The simplest gates in the PASM are the two-input AND gate, the two-input NOR gate, the inverter, the buffer and the XOR gate. They range in size from 2 to 6 MOS transistors. The BALLISTIC-generated layouts of three of these gates are illustrated in Figure 4.1. A circuit for the XOR gate is shown in Figure 4.2(a) [Rab95]. Its corresponding layout is given in part(b) of the figure.

The layout area and signal propagation delays (average delay for passing 0 and 1) for each of these gates are shown in Table 4.1.

<table>
<thead>
<tr>
<th>Gate</th>
<th>Area ($\mu m^2$)</th>
<th>Delay (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>99.6</td>
<td>0.10</td>
</tr>
<tr>
<td>NOR</td>
<td>68.1</td>
<td>0.14</td>
</tr>
<tr>
<td>NOT</td>
<td>56.4</td>
<td>0.08</td>
</tr>
<tr>
<td>buffer</td>
<td>78.0</td>
<td>0.21</td>
</tr>
<tr>
<td>XOR</td>
<td>203.4</td>
<td>0.10</td>
</tr>
</tbody>
</table>

Table 4.1: Gate area and average propagation delay.

**SRAM cell**

The SRAM cell circuit that is used in the PASM is taken from [Kav98]. As shown in Figure 4.3(a), it is a five transistor circuit that consists of two cross-coupled inverters and one NMOS pass-transistor for programming. Having two same-strength inverters...
Figure 4.3: SRAM cell schematic and layout based on two cross-coupled inverters.

would make the gate unstable when being programmed because neither would be able to dominate and force the voltage to a defined level. Hence, the feedback inverter is built with weak transistors. The SRAM cell area is 101 \( \mu \text{m}^2 \).

**Two-input multiplexer**

Two-input multiplexers are used often in the PASM and can therefore have a significant impact on its total area. We first considered using a full pass-transistor logic circuit, shown in Figure 4.4(a). However, because the PASM uses many (19) of these multiplexers, the smaller design, shown in part(b) of the figure, is used. The circuit passes logic 1's at only \( V_{dd} - V_{Tn} = 2.6V \), using a 3.3V power supply, but this drawback is ignored in favour of saving chip area. The layout of the multiplexer in part(b) is shown in part(c) of the figure.

Figure 4.4: Two-input multiplexer designs.

**Four-input AND gate**

The largest gate in the PASM is the four-input AND gate located at the bottom of
Figure 3.8. Since it has 4 NMOS transistors in series, the gate switches rather slowly, with an average propagation delay of 0.55 ns. This is not important because the AND gate is not in the critical path of the PASM. The layout of the four-input AND gate is shown in Figure 4.5(a), and its area is 150 $\mu$m$^2$.

**Bit-select gate**

True and inverted forms of both the $x$ and $y$ input bits must be available at each bit-slice. One way to achieve this would be to use an XOR gate with one input acting as the inversion control. Another method, using an inverter and a multiplexer was also examined and proved to be more area efficient than an XOR gate. This inverter-multiplexer combination is called the bit-select, or *bsel* circuit in the PASM architecture. Figure 4.5(b) shows the layout of a *bsel* circuit. The area of the gate is 136 $\mu$m$^2$, compared to 203.4 $\mu$m$^2$ for an XOR gate. The average propagation delay is 0.12 ns.

**Carry Logic**

The carry logic circuit is responsible for generating the appropriate carry-out signal given the carry-in, propagate and generate signals. The logic equation for the carry-out signal is given by

$$c_{i+1} = g_i + p_i - c_i$$

Figure 4.6(a) shows how this logic equation could be implemented in a PASM bit-slice using an AND gate and an OR gate. Six MOS transistors are needed for both types of gates giving a total of twelve transistors for this implementation. If we reduce the
Figure 4.6: Various implementations of the carry logic circuit.

circuit to NAND gates, as shown in part(b) of Figure 4.6, we can reduce the cost of the circuit to 6 MOS transistors. Each NAND gate requires 4 transistors, but we save 2 by reducing the AND gate that generates \( g_i \) to a NAND gate. The average propagation delay of this circuit is 0.18 ns. Part(c) shows a third implementation [Lew96] of the carry circuit which also costs 6 transistors. Note that the carry-in signal is inverted coming into the bit-slice and must be inverted to correctly generate the \( sum \) bit. The average propagation delay of this circuit is 0.1 ns, and because it is faster than the circuit of part(b), we chose it as the PASM carry logic circuit.

D flip-flop
The D flip-flop is negative edge-triggered with asynchronous reset. A schematic of the 6 NOR gate [CHZ97] design is shown in Figure 4.7. Its layout implementation is shown in Figure 4.8. The flip-flop layout consists of 26 transistors and has an area of 627 \( \mu m^2 \). The average clock-to-output propagation delay is 1.14 ns.

Four-to-one multiplexer
As mentioned in Section 3.1, we decided to multiplex the bit-slice output pins with four distinct logic signals. Each bit-slice can output any of the \textit{propagate}, \textit{generate}, adder output, or PASM-wide comparison signals via a four-to-one multiplexer. The
A multiplexer, shown in Figure 4.9, is based on NMOS pass-transistors with inverters providing the true and complemented forms of the two control bits. The control signals come from SRAM cells labeled SEL0 and SEL1 in Figure 3.8 and connect to the gates of the two inverters on the right of Figure 4.9. A mapping of the multiplexer output as a function of SEL0 and SEL1 appears in Table A.2. The multiplexer area is 287 $\mu$m$^2$ and the average propagation delay is 0.1 ns.

### 4.2.2 Bit-Slice Layout

A bit-slice of a PASM is shown in Figure 4.10. As mentioned in Section 3.2, data flows from left to right. There are two power wires, one at the top and one at the
bottom, and a single ground wire running through the middle of the circuit. The power-ground/ground-power configuration forces all gates on the lower half of the circuit to be flipped vertically. A total of 102 transistors are used to implement a bit-slice.

The pattern of gates is very close to that of the block diagram in Figure 3.2. This is done to preserve the flow of data from left to right. Notably, the flip-flop is located in the top right corner of the bit-slice layout, while the four-input multiplexer is located close to the middle on the lower half. These are the two largest components in the bit-slice layout.

Due to the large number of power and ground connections, a few polysilicon wires are used to make connections between gates. Since we anticipated that the metal2
and metal3 levels would eventually become a bottleneck, we decided to refrain from using them at the bit-slice level unless absolutely necessary.

The dimensions of a single PASM bit slice are 100 μm (horizontal) × 35 μm giving an area of 3500 μm².

Latch-Up

Latch-up can be a problem in CMOS technologies. It occurs when the wells and substrates form parasitic n-p-n-p structures which, when stimulated by current spikes, can cause shorting of power lines. Latch-up can be avoided by liberally adding well and substrate contacts to the layout design. [WE94] suggest that contacts be added every 5-10 transistors or every 25 μm in any design, which is provided in our design.

Electromigration

Electromigration, which is the transporting of metal ions over time causing an eventual short or breakage, is a potentially serious problem. Power wires, which carry a DC current for an extended period of time, are especially vulnerable. General signal wires are not as susceptible given that their voltage level changes frequently. Electromigration can be avoided if power and ground wires are properly sized such that the current density in these wires is not high.

[Rab95] suggests that power and ground wires be large enough that the current is less than 0.5 to 1 mA/μm. The PASM has ground and power widths of 2.4 μm, and simulations show a current of 0.1 mA. Hence, the layout is within the suggested limits for protecting against electromigration.

Timing

Since the bit-slice design does not incorporate any gates with large fan-out, there are no large capacitances to drive which would negatively impact the speed-performance of a PASM bit-slice. The critical path from input to output passes through 7 gates and the high-to-low propagation delay, \( t_{pHL} \), is 1.8 ns. The low-to-high propagation delay, \( t_{pLH} \), is approximately 3 ns. A Spice graph of the delay between the \( x \) and \( y \) inputs changing and the resulting change in the output signal is shown in Figure 4.11. The PASM block is configured to compute \( x - y \), with a carry-in of 0.
Output Driver

PASM bit-slice outputs not only have to drive the inputs of adjacent PASM blocks, but also have to drive local and chip-wide interconnect wires. To deal with the heavy capacitive loading of interconnect wires, output drivers are needed. [Rab95] states that the optimal scaling factor for cascaded inverters is dependent on the capacitive load to be driven, and is slightly larger than $e$ (2.781). We used two series-connected inverters, scaled by a factor of 3. The schematic for the output driver is shown in Figure 4.12. It has tri-state capability, which is used to make a programmable connection to interconnect wires. If the SRAM cell labeled drive stores a 1, the driver is turned on, otherwise, the driver output is tri-stated. The layout of two output drivers appears in Figure 4.13. Since the PASM block is nibble-controlled, a single SRAM cell is sufficient to program the four output drivers. It is located in the middle of the layout of Figure 4.13.

4.2.3 PASM Block Layout

A complete PASM block consists of 4 bit-slices stacked vertically, together with some extra logic and 9 SRAM cells. Figure 4.14 gives a detailed view of the layout. Bit-slice output drivers are not included in this figure.

Seven SRAM cells are located at the top of the circuit. The eighth and ninth SRAM cells are at the bottom. The four-input AND gate is also located at the bottom of the layout, along with the NOR gate, a 2-input multiplexer and the three
bit-select units. These gates are responsible for generating the EQ, LT and GT signals along with the carry-out signal.

The EQ, GT and LT signals are multiplexed, one each, on the output pins of the lower three PASM bit-slices. The top bit-slice has its unused 4:1 multiplexer input connected to $V_{dd}$. The connections between the EQ, GT and LT signals and the 4:1 multiplexers are in metal2. The control lines for the 4:1 multiplexers come from two SRAM cells, one at the top of the circuit and one at the bottom. These signals are also in metal2.

As previously mentioned, the 4-input AND gate takes as input the XOR of the input-bit pairs of all four bit-slices. Since the AND gate is located at the bottom of the circuit, its inputs are wired in metal2.

The dimensions of the PASM block are 123 $\mu$m (horizontal) $\times$ 186 $\mu$m, giving an area of 22878 $\mu$m$^2$. A total of 505 transistors are used. The complete layout has been simulated using HSpice and tested successfully.

![Figure 4.12: Schematic of an output driver.](image)

Figure 4.13: Layout of an output driver.
4.3 Routing Multiplexer

In the CFA, the chip-wide interconnect wires connect to the PASM block inputs via multiplexers. Another option would have been to use pass-transistor switches in the interconnect. However, this method requires that a separate SRAM cell be included for each switch, which would make the total area of the design much larger than one that uses multiplexers.

The size of the routing multiplexers is an important issue. For our first attempt
at building a PASM cluster, one objective is to ensure that sufficient interconnect resources are provided to be realistic for a useful CFA chip. At the same time, an excessive number of multiplexer connections should be avoided, because this could negatively impact speed performance. We chose to use a 16:1 multiplexer, which needs 4 SRAM cells. The routing multiplexer is an NMOS pass-transistor circuit and is shown in Figure 4.15.

The layout of the routing multiplexer is shown in Figure 4.16. Power and ground wires are at the top and bottom, respectively. Two SRAM cells are located at both
the left and right sides of the multiplexer. The SRAM cell outputs are buffered to ensure that the pass-transistor gates of the multiplexer are properly driven. The multiplexer design uses metal2 and metal3 only for input and output contacts in an attempt to conserve these two metal layers for future use. The maximum simulated propagation delay through the multiplexer is 0.55 ns and its area is 3641 \( \mu \text{m}^2 \).

### 4.4 PASM Tile and Routing Multiplexers

Figure 4.17 shows the layout of a PASM with its 10 routing multiplexers. A minimal amount of metal2 and metal3 is used, in an attempt to save these layers for use in interconnect wiring. Notice that there are six routing multiplexers on the left side, and four on the right. The two leftmost multiplexers are for the carry-in and multiplier control inputs. The remaining four are connected to the \( x \) data inputs. The four multiplexers on the right side are connected to the \( y \) data inputs. The PASM core logic is located in the middle, with bit-slice output drivers above and below.

The dimensions of the PASM block are 252 \( \mu \text{m} \) (horizontal) \( \times \) 246 \( \mu \text{m} \), for a total area of 61992 \( \mu \text{m}^2 \). The PASM core makes up 40% of the total area, the output drivers and switches make up 3%, and the remaining 57% is taken up by the routing multiplexers.
4.5 Area Estimations

A technology mapper has been developed [Kav98] which enables us to map circuits from a Hardware Description Language (HDL) to a netlist of PASMs. In effect, we know how many PASMs are needed to implement the HDL application and how to connect them together. Given a netlist of PASMs, we can compare the amount of silicon needed to implement a set of benchmark applications on our architecture and on commercial devices.

Table 4.2 shows a set of simple benchmark circuits and the area that is needed to map them into PASMs, general purpose FPGAs and ASICs. The PASM area numbers were obtained by creating a target library in Synopsys, then mapping the benchmark circuits into that library. The area of a PASM tile, which we know
Table 4.2: Area comparisons of various architectures on arithmetic benchmark circuits from our layout, is taken as the area of a PASM block, plus the area of its 10 routing multiplexers. The total is 65 kμm². The numbers for the ASIC column were obtained by mapping circuits into the target standard cell library Hewlett Packard CMOS14 using Synopsys. [Pac97] then recommends adding 25 percent to the results to account for the routing area in standard cells. This method of estimating area of standard cell designs was suggested in the Hewlett-Packard reference manual for the CMOS14 target library. The FPGA area numbers were similarly derived. We mapped circuits to the Altera Flex10K architecture using Maxplus+II and Synopsys to determine the number of LUTs used by that circuit. Using area numbers for a LUT that were provided to us by industry [Pac97], we were then able to determine the amount of silicon needed to implement the same circuit using LUTs in the CMOS14 process. The number used for the area of a 4-LUT is 45 kμm².

As seen in the table, the CFA can efficiently implement arithmetic structures such as adders and multipliers. It is approximately 6 times less area efficient than ASICs, while conventional FPGAs are more than 22 times less efficient than ASICs. The CFA uses on average 4 times less silicon than the conventional FPGA on our bench-
mark circuits and can therefore be a viable alternative for implementing arithmetic structures.

The next chapter discusses issues pertaining to the layout of a complete PASM *cluster*, including the design of the *cluster-level interconnect*.
Chapter 5

PASM Cluster and Interconnection Architectures

This chapter discusses issues involved in the design and implementation of a PASM cluster and its interconnection resources.

5.1 Cluster Architecture

An illustration of a CFA chip is shown in Figure 5.1. It consists of two main logic resources. The PASM logic blocks are arrayed in groups called PASM clusters. The number of PASMs that should be included in a cluster, and the way in which these PASMs should be interconnected is an open issue. In addition to PASM clusters, the figure also shows a LUTB cluster. A LUTB is a group of LUTs with local interconnect for wiring the LUTs together. The number of LUTBs that should be included in a LUTB cluster is also an open question. Although Figure 5.1 indicates a single LUTB cluster in the chip, future research may show that multiple LUTB clusters are more attractive. The LUTB clusters and PASM clusters together are called a section, and the interconnection wires within the section are called the section-level interconnect. A complete CFA chip may comprise one or more sections, with appropriate interconnection wiring.

Since the goal of this thesis is to investigate layout issues for the CFA, we need
to decide how much of an entire chip should be fabricated at this time. A sufficient portion of the chip must be included so that meaningful testing can be done, and yet limit the scope of the work to provide a good chance of success. We decided that a reasonable compromise is to fabricate one PASM cluster. Assuming that this is successful after testing, it is not difficult to later fabricate multiple PASM clusters in combination with a LUTB cluster to create a section.

Given the goal of laying out a PASM cluster, we need to decide on the number of PASM to include within the cluster, and how to interconnect them. These decisions are discussed in this chapter, and the final layout is presented.

![Figure 5.1: CFA architecture.](image)

### 5.1.1 Cluster Size

For the implementation in this thesis, we chose a cluster size of 16 PASMs, in which we can implement modestly sized arithmetic structures such as 64-bit adders. For future consideration, the cluster layout code is written such that the number of PASMs in a cluster can be easily changed by altering a loop index variable in the BALLISTIC code.

### 5.1.2 Cluster Shape

Using the preliminary results of the layout, and with a desire to make the cluster layout balanced on all three metal levels, we decided to arrange the 16 PASM tiles in an 8 row by 2 column configuration. This decision takes into account the need to include a number of section-level interconnect wires at a later point. The final
size and shape of a PASM cluster are shown in Figure 5.2. The cyan boxes on the periphery represent shift registers used to program the PASM cluster. They are discussed in Section 5.4.

5.1.3 Routing Switches

Each of the routing multiplexers for the 10 inputs to a PASM connects to the cluster-level interconnect wires via a set of switches. As described in Section 4.3, each multiplexer has 16 inputs, for a total of 160 possible inputs. However, some of the inputs to each multiplexer are used for dedicated connections, as described in Section 5.2. This number of dedicated connections is small, and never exceeds 4 for any multiplexer, leaving a minimum of 12 wires per multiplexer to be connected to the cluster interconnect wires.

The arrangement of routing switches can have a large impact on the routability of an architecture [ZWC93]. If the switches are poorly placed, many signals may attempt to use a small number of wires, leading to an architecture that is unroutable for many circuits.

The switch placement for a single PASM is based logically on the arrangement
shown in Figure 5.3. The switches are arranged such that each of the four bits of an input vector can connect to the cluster-level wires with minimal interference from the other three input vector bits. For example, the \( x_0 \) input connects to wire 1, \( x_1 \) to wire 2, \( x_2 \) to wire 3 and \( x_3 \) to wire 4. The second connection of \( x_0 \) is to wire 5, \( x_1 \) to wire 6 and so on. The \( y \) inputs follow a similar pattern.

## 5.2 PASM Cluster-Level Interconnect

The cluster-level interconnect permits data flow inside the PASM cluster. Many possible solutions exist to the problem of designing the interconnect of field programmable devices. As discussed in 2, different commercial architectures have adopted different strategies, varying from hierarchical designs to segmented designs. Similar options exist for the cluster-level interconnect.

Figure 5.4 shows a PASM with its 10 routing multiplexers in grey. To avoid clutter in the figure, the connections from the multiplexers' outputs to the PASM inputs are not shown. Instead, the label to the left of each multiplexer indicates
which PASM input it feeds. A dot on a multiplexer indicates an electrical connection to a multiplexer input. Each multiplexer has 16 inputs. The inputs in Figure 5.4 are connected to an interconnection network of unknown size and shape. This figure acts as a template on which we will describe the final interconnect architecture.

One solution to the interconnect problem would be to design the interconnect in a bit-controlled style, similar to the Xilinx XC4000 series interconnect, with large busses of undedicated wires connecting PASMs. This is a simple solution and it offers good flexibility for moving and shifting data.

Another possibility is to take advantage of the fact that the PASM architecture is based on 4-bit operands. The data inputs are two 4-bit vectors, and the output is also a 4-bit vector. An intuitive design of the cluster interconnect would be to move data in 4-bit chunks, on 4-bit wide busses. This type of design is called nibble-controlled interconnect.

A nibble-controlled interconnect has the advantage that the routing multiplexers for the $x_0, x_1, x_2$ and $x_3$ inputs are all programmed to connect to a single 4-bit bus. As a result, only 4 SRAM cells are needed to program all 4 routing multiplexers, compared to 4 SRAM cells per multiplexer. For a PASM, this would translate into a significant savings of 24 SRAM cells. A disadvantage of a nibble-controlled
interconnect is that the flexibility of individually programming each PASM input is lost. As well, the control of this type of interconnect is more complex than that of a bit-controlled interconnect. The investigation of a nibble-controlled interconnect is left as future work.

We know that there are certain cases in which data will be passed from one PASM to another inside a cluster with a high degree of regularity. For example, the case where a PASM passes its 4-bit output vector to the PASM immediately below is anticipated to be common. Hence, it is logical to make dedicated connections, or fixed wires, from a PASM to its lower neighbour.

A similar case involves implementing shift registers. It would be cumbersome to make shift registers by performing the data shifting in the interconnect wires. A more efficient implementation would be to feed the output bits of a PASM back to its own multiplexers, shifted by one position in both directions. No cluster-level interconnect wires are used, and a simple means of bit shifting is provided. Figure 5.5 shows how a PASM could implement a left-to-right shift register. Only one input in the routing multiplexers of the $y$ input is needed, along with the area used by the feedback wires.

Multiplication, as implemented in PASMs, also involves bit shifting. In each step, the summand is added to the partial product. To accomplish this in a PASM, the summand must be shifted by one bit position to the right. As previously mentioned, shifting data in the interconnect can be inefficient. We therefore decided to sup-
Figure 5.6: Dedicated wires between two PASMs in a multiply.

Port this type of shifting for multiplication with dedicated wires between vertically adjacent PASMs. Also, since the least significant bit of the upper summand is not used, and the remaining three bits are shifted to the right, the most-significant bit comes from the carry-out bit of the upper summand. This too, is supported with a dedicated connection.

Figure 5.6 demonstrates the need for dedicated connections in a typical multiplication example. Part (a) of the figure shows the decomposition of a 4 × 4 multiply into its four summands, with the resulting product at the bottom. Part (b) shows how summands 1 and 2 are implemented in adjacent PASMs, and the dedicated connections needed to efficiently support the multiply. Notice that the outputs of the top PASM are shifted to the right, with the carry-out bit supplying the most-significant data bit to the lower PASM. The least-significant bit is sent onto a general interconnect wire where it will eventually form a part of the final product.
5.2.1 Architecture of the Cluster-Level Interconnect

The final architecture of the cluster-level interconnect is a combination of bit-controlled and dedicated wires. Figure 5.7 shows how the cluster-level interconnect looks for two vertically adjacent PASMs.

For ease of understanding, only the multiplexer input connections for PASM 1 are shown in the figure. As well, only the multiplexers for the data input vectors $x$ and $y$ are shown. Dark blue lines indicate inputs to multiplexers that come from bit-controlled wires. Note that to avoid clutter in the figure, dark blue lines appear to be shared by 4 multiplexer inputs. In our architecture, each multiplexer input joins the cluster interconnect by a unique wire. Each $x$ input multiplexer connects to 15 interconnect wires, while each $y$ input multiplexer connects to 12 interconnect wires. Red lines indicate feedback wires from a PASM output to its inputs. These wires are used to implement shift registers. This is only done on the $y$ inputs to
conserve inputs to the $x$ multiplexers. Shifting can be done either to the right or left. Green lines indicate PASM outputs that are sent to the immediately lower PASM as inputs. Both the $x$ and $y$ inputs can be connected in this way. The $y$ input also gets shifted input data to facilitate multiplication, as demonstrated in Figure 5.6. The $x$ input does not get shifted data because the PASM architecture is not symmetrical for multiplication. Lastly, the cyan line represents the carry-out bit of the immediately upper PASM. It is a dedicated connection needed for multiplication, but only feeds the $y_3$ input multiplexer. The carry-out bit is also brought as a dedicated input to the carry-in multiplexer, but this is not shown in Figure 5.7.

The PASM outputs can also drive the cluster interconnect wires, depending on the value stored in the drive SRAM cell. A mapping of the PASM outputs to the cluster interconnect wires is given in Table A.5.

### 5.3 Layout of Cluster-Level Interconnect

The cluster-level interconnect wires are implemented in metal3. The number of wires is largely dictated by the amount of space left for metal3 in the PASM design. When we were deciding what shape to make a cluster, one of the motivating factors was the

![Figure 5.8: PASM tile floorplan.](image-url)
need to have approximately 80 cluster-level interconnect wires. We thought that the width of the 8 row by 2 column shape would give us roughly this number, and it turns out that we are able to put in 38 per PASM, or 76 wires in total. The cluster-level wires run horizontally, as indicated in Figure 5.8.

The cluster-level wires are spaced more than the minimum specified distance for the HP CMOS14 process. Average wire separation is 35% above the minimum required distance.

The orthogonal direction (vertical in Figure 5.8) is reserved for section-level interconnect wires using metal2. These wires are used to share data between different clusters, or between LUTBs and clusters. The number of wires and the switch patterns for section-level interconnect is left as future work, although we estimate that we can fit approximately 40 wires per PASM, or 320 section-level wires.

A picture of the three metal levels in a PASM is shown in Figure 5.9. Metal levels 1 and 3 are largely filled, while space was deliberately left in the metal2 level to accommodate the section-level interconnect. We have attained a good level of balance between metal3 and metal1 thus far, and anticipate that the section-level interconnect will fill in the gaps in metal2.

The physical layout of the switches discussed in Section 5.1.3 is shown in Figure 5.10. The difference between it and the logical pattern of Figure 5.3 is that the physical pattern is affected by the location of the routing multiplexers relative to the

(a) Metal 1  (b) Metal 2  (c) Metal 3

Figure 5.9: Metal levels in the PASM tile.
Figure 5.10: Cluster-level switch physical locations.

PASM tile. The cluster-level interconnect wires run horizontally, while the routing multiplexers are arranged as shown in Figure 5.8. Consequently, it is more natural to connect the lower level of multiplexers \((x_2, x_3, c_{in}, y_2, y_3)\) to the bottom half of the interconnect tracks. The logical pattern is preserved if the routing software understands that the physical wire locations still represent the intended logical pattern. Leftover switches are added in a random fashion, which previous studies [Lem00] have shown does not negatively affect the routability of the architecture.

Connections between multiplexer inputs and the interconnect switches are done in metal2. The wiring pattern for a section of two PASM blocks is demonstrated in Figure 5.11. For clarity, only metal2 (purple) and metal3 (cyan) are shown in the figure. Notice that the majority of metal2 runs perpendicularly to the metal3 interconnect wires. Quite often, however, we had to use any available gaps in the metal2 and metal3 levels to successfully route connections, requiring some horizontal metal2 wires. Any horizontal gaps in the metal2 level are reserved for future section-level interconnect wires.

Each multiplexer has, on average, two inputs connected to the cluster wires of the vertically opposite PASM. This accounts for the metal2 wires that cross the large horizontal gap in the middle of Figure 5.11. Note that multiplexer inputs used for dedicated interconnect wires are not included in the figure.
5.4 SRAM Cell Programming

Programming the 50 SRAM cells in each PASM is an issue that is dependent on the cluster shape, because of the constraints imposed by the layout on where metal lines can be placed. An SRAM cell is programmed when its control wire is asserted,
causing the SRAM cell to latch in the data value present on its \textit{data} wire. Using a two-dimensional grid of control and data lines, we can efficiently program any number of SRAM cells, where the number of SRAM cells is equal to the number of grid intersection points. Shift registers drive the data onto the control and data lines.

A routing multiplexer, which contains 4 SRAM cells, is used to demonstrate this method of SRAM cell programming \cite{CSC+93}, in Figure 5.12. When control line \textit{pgm-cntl1} is active, both SRAM cells 1 and 2 will latch the data present on the \textit{pgm-data} line. Similarly, SRAM cells 3 and 4 are programmed when the \textit{pgm-cntl2} line is active. Both \textit{pgm-cntl} lines may be active at the same time, but only if the SRAM cells sharing a common \textit{pgm-data} wire are to latch the same data. This type of scheme can be extended to any number of SRAM cells by increasing the number of control and data lines.

We need to extend this idea for the 50 SRAM cells in each PASM tile. Ideally, we would like to have an equal number of control and data lines so that each type of line drives an equal number of SRAM cells. In our cluster design with 800 SRAM cells, this would translate into 29 control and data lines, with 29 SRAM cells hanging off each type of line. However, our cluster is not symmetrical. It is arranged as 8 rows of 2 columns of approximately square PASMs. We could comfortably run 29 lines vertically since we have 8 PASM heights to use. But 29 lines across 2 PASM widths would not leave any room for the number of section-level interconnect lines that we
anticipate needing. As a result, we decided to run 20 control lines horizontally and 40 data lines vertically. On a per PASM basis, this means 10 data lines and 5 control lines.

Figure 5.13 shows how the 50 SRAM cells in a PASM are connected to the five data (pgm-data) lines. Each of the 10 SRAM cells that are connected to a pgm-data line latches in the data on that line subject to the assertion of their pgm-cntl lines, similar to the method shown in Figure 5.12.

The 5-bit shift register used to drive data onto the pgm-data lines appears in

Figure 5.14: Two bits of the 5-bit shift register.
Figure 5.15: Mapping of SRAM cells to 10 control lines.

Figure 5.14. For clarity, only 2 of the five bits are shown. The register is built with negative-edge-triggered D-type flip-flops, and has buffered outputs. The buffering is quite strong to enable the register to properly drive the inputs of ten SRAM cells. Note that since the register will be aligned horizontally, all long connections are in metal3. The pgm-data wire contacts are in metal2 to match the PASM data wires.

Figure 5.15 shows the mapping of the control (pgm-cnt1) lines to the SRAM cells in the PASM. Each pgm-cnt1 line controls five SRAM cells. A positive pulse on a control line causes the SRAM cells that are connected to it to latch whatever data is present on their respective pgm-data lines at that time.

The layout of the 10-bit shift register is shown in Figure 5.16. Similar to the picture of the 5-bit register, only 4 of the 10 bits are shown. The 10-bit register is essentially two 5-bit registers back-to-back and since it will be aligned vertically, all long wires are in metal2. The outputs are also strongly buffered since each drives 40 small transistor gates. The pgm-cnt1 contacts are in metal3 to match the PASM control wires.
Figure 5.16: Four bits of the 10-bit shift register.

### 5.5 Cluster Layout

The layout and floorplan of a PASM cluster are shown in Figure 5.17. The programming control shift registers are located on the left of the figure, while the programming data registers run along the top and bottom. The horizontal metal3 wires represent the 76 cluster-level interconnect wires. The metal3 wires running across the middle of the cluster represent dedicated interconnect wires between adjacent PASMs. There are two power wires, one above and below the two levels of PASMs. A ground wire runs in the middle.

The dimensions of the design are 2260 µm x 685 µm, giving an area of 1.55 mm². More than 22000 MOS transistors are included in the cluster layout.

Figure 5.18 shows a more detailed view of a part of the cluster. Only the three metal levels are shown to demonstrate the wiring patterns. Dedicated connections between PASMs run horizontally through the middle of the layout, as shown in both Figure 5.17 and Figure 5.18.

#### Clock Distribution in a Cluster

With 16 PASMs in a cluster, and 4 flip-flops per PASM, the number of clocked elements in a cluster is 64. And given that the PASM cluster is over 1.5 mm in length, clock distribution is an important issue. We used an H-tree design, shown in Figure 5.19, to minimize clock skew between PASMs. The global reset line uses a similar H-tree pattern.
Timing

Figure 5.20 shows an HSpice graph of the four outputs of a single PASM in a cluster. The PASM is configured as an adder. The only externally generated signals are the cluster interconnect wires, the clock signals, reset signals and power rails. The four $x$ and $y$ data inputs are logic 0's until $t = 1400 \mu s$, at which point both $y_1$ and $y_2$ become logic 1's. Both inputs return to logic 0's at $t = 1800 \mu s$. The average propagation delay between driving a cluster interconnect wire and the PASM output
is approximately 4 ns. While the cluster SRAM cells are being programmed, the PASM outputs are subject to fluctuations, which can be seen at 250 ns and again at 800 ns. These fluctuations disappear once the cluster is fully programmed.
Figure 5.20: Output waveforms of a single PASM in a cluster.
Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis has discussed the design and implementation of a full-custom layout of a CFA cluster. The layout was done in a 0.5 μm CMOS process with three metal levels. The lowest level of interconnect in the CFA architecture, called the cluster-level interconnect, was also designed and implemented.

The design was synthesized using the BALLISTIC layout language and has been functionally verified using HSpice.

One of the motivations of this work is to prove or disprove the notion that the CFA is more area-efficient at implementing arithmetic structures than a conventional FPGA. We have shown that on a set of benchmark circuits, the CFA is on average 4 times more area efficient than a conventional FPGA.

6.2 Future Work

We intend to fabricate and test a PASM cluster. The results of testing will dictate what type of changes need to be made to the design of the PASM layout.

A future goal of this project is to assemble and fabricate the complete CFA architecture. This involves completing the design of the LUTB blocks as well as the design and implementation of the top-level CFA interconnect.
This work focused strictly on minimizing the area of the cluster layout and consequently, it is likely that the speed performance of the circuit can be improved. We will investigate this in future work.
Appendix A

CFPA Programming

This appendix contains programming information for a PASM cluster. Specifically, SRAM cell programming maps for each multiplexer are included. Maps of the connections between interconnect wires and routing multiplexer inputs are also provided.

Table A.1 shows the mapping of the 10 SRAM cells in the PASM core logic. Refer to Figure 3.8 for PASM architecture details.

Table A.2 shows how the control SRAM cells SEL0 and SEL1 map to the 4-input multiplexer in each PASM bit-slice. Refer to Figure 3.2 for bit-slice architecture details.

A PASM cluster is divided into 8 rows of 2 columns of PASMs. The 76 cluster-level interconnect wires span the width of the two PASM columns, as described in Section 5.3. Each of the two PASMs in a cluster row connects to a subset of these 76 interconnect wires via a routing multiplexer. Tables A.3 and A.4 show how the inputs of the ten routing multiplexers in each PASM connect to interconnect wires. Note that a wire labeled outx indicates that the multiplexer input is connected to bit-slice output x of the same PASM.

The outputs of a PASM cluster can drive the interconnect wires, or they can be tri-stated. The mapping of the outputs to the interconnect wires is given in Table A.5.
<table>
<thead>
<tr>
<th>SRAM Name</th>
<th>Function</th>
</tr>
</thead>
</table>
| XSEL      | controls selection of $x$ input  
1 → $x$
0 → $\overline{x}$ |
| AS/M      | controls selection of $m$ input  
1 → $m$
0 → $Vdd$ |
| OS        | determines output source  
1 → combinational output
0 → registered output (flip-flop) |
| S/A       | controls $y$ input  
1 → $\overline{y}$
0 → $y$ |
| ADDSUB    | determines which bit will control $y$ input  
1 → $S/A$
0 → $m$ |
| CCNTL     | determines source of carry input  
1 → $S/A$
0 → carry - in |
| INVC      | controls LT,EQ,GT outputs  
1 → $LT$, $EQ$, $GT$
0 → $LT$, $\overline{EQ}$, $\overline{GT}$ |
| SELO      | refer to table A.2 |
| SEL1      | refer to table A.2 |
| DRIVE     | 1 → drive interconnect
0 → tri-state |

Table A.1: SRAM programming code for PASM
<table>
<thead>
<tr>
<th>SEL0</th>
<th>SEL1</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>generate ((x \cdot y))</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>propagate ((x \oplus y))</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>comparison ((GT, LT, EQ))</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>bit slice output ((x \oplus y \oplus carry))</td>
</tr>
</tbody>
</table>

Table A.2: SRAM programming code for 4-input PASM multiplexer

Figure A.1: Map of cluster wires to multiplexer inputs.
Table A.3: Map of switches to cluster wires for left-column PASMs. * indicates that the multiplexer input connects to either the output or carry signal of the previous PASM in the cluster. Cluster wires labeled out that are not accompanied by * indicate outputs that come from the same PASM.
Table A.4: Map of switches to cluster wires for right-column PASMs. * indicates that the multiplexer input connects to either the output or carry signal of the previous PASM in the cluster. Cluster wires labeled out that are not accompanied by * indicate outputs that come from the same PASM.
Table A.5: PASM outputs connect to the cluster wires listed above. Refer to Figure 2.1 for map of PASMs relative to interconnect wires.

<table>
<thead>
<tr>
<th>PASM</th>
<th>Output 1</th>
<th>Output 2</th>
<th>Output 3</th>
<th>Output 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>38</td>
<td>37</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>36</td>
<td>35</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td>34</td>
<td>33</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>8</td>
<td>33</td>
<td>31</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>10</td>
<td>30</td>
<td>29</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>12</td>
<td>28</td>
<td>27</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td>14</td>
<td>26</td>
<td>25</td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td>16</td>
<td>24</td>
<td>23</td>
</tr>
<tr>
<td>8</td>
<td>76</td>
<td>75</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td>9</td>
<td>74</td>
<td>73</td>
<td>41</td>
<td>42</td>
</tr>
<tr>
<td>10</td>
<td>72</td>
<td>71</td>
<td>43</td>
<td>44</td>
</tr>
<tr>
<td>11</td>
<td>70</td>
<td>69</td>
<td>44</td>
<td>46</td>
</tr>
<tr>
<td>12</td>
<td>68</td>
<td>67</td>
<td>47</td>
<td>48</td>
</tr>
<tr>
<td>13</td>
<td>66</td>
<td>65</td>
<td>49</td>
<td>50</td>
</tr>
<tr>
<td>14</td>
<td>64</td>
<td>63</td>
<td>51</td>
<td>52</td>
</tr>
<tr>
<td>15</td>
<td>62</td>
<td>61</td>
<td>53</td>
<td>54</td>
</tr>
</tbody>
</table>
Bibliography


