

©Copyright 2005

Mark Holland

Automatic Creation of Product-Term-Based Reconfigurable Architectures for
System-on-a-Chip

Mark Holland

A dissertation submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

University of Washington

2005

Program Authorized to Offer Degree:
Department of Electrical Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Mark Holland

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of Supervisory Committee:

Scott Hauck

Reading Committee:

Carl Ebeling

Scott Hauck

Larry McMurchie

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of the dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Proquest Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature_____

Date_____

University of Washington

Abstract

Automatic Creation of Product-Term-Based Reconfigurable Architectures for
System-on-a-Chip

Mark Holland

Chair of the Supervisory Committee:

Professor Scott Hauck

Department of Electrical Engineering

Technology scaling has brought the IC industry to the point where several distinct components can be integrated onto a single chip. Many of these System-on-a-Chip (SoC) devices would benefit from the inclusion of reprogrammable logic on the silicon die, as it can add general computing ability, provide run-time reconfigurability, or be used for post-fabrication modifications. Also, by tailoring the reconfigurable fabric to the SoC domain, additional area/delay/power gains can be achieved over current, more general fabrics. Developing a domain-specific reconfigurable fabric has traditionally taken too much time and effort to be worthwhile. We are alleviating these design costs by automating the process of creating domain-specific reconfigurable fabrics: a project we call Totem.

This dissertation details our work in creating tools that will automate the creation of domain-specific PLAs, PALs, and CPLDs for use in SoC devices. The input to the toolset is a group of circuits that need to be supported on the reconfigurable fabric. The tools then create a PLA, PAL, or CPLD that is tailored to the specific test circuits, with an option for strategically providing additional resources in order to support future, unknown circuits. The output of the toolset is a fully optimized VLSI layout of the reconfigurable fabric. This VLSI layout can be provided to a designer for direct integration into an SoC design.

Our domain-specific CPLD architectures based on full crossbars outperform representative fixed architectures by 5.6x to 11.9x in terms of area-delay product. Our toolset has also been used to find several more efficient fixed architectures, but our domain-specific architectures still outperform these new fixed architectures by 1.8x to 2.5x. Sparse-crossbar-based CPLDs have also been created, and require only 37% of the area and 30% of the propagation delay of the full-crossbar-based CPLDs. Lastly, an analysis of our sparse-crossbar-based CPLD architectures suggests that, in order to support future circuits, the crossbar switch density should be augmented by 5% over the base density, and additional PLAs of the base PLA-size should be provided for additional logic utilization.

Table of Contents

List of Figures	iv
List of Tables	ix
1 Introduction.....	1
2 Programmable Logic Devices.....	5
2.1 Programmable Logic Devices.....	5
2.1.1 Complex Programmable Logic Devices (CPLDs).....	12
2.1.2 Case Study – Xilinx CoolRunner XPLA3 CPLD.....	13
3 CAD for Programmable Logic.....	17
4 Reconfigurable Hardware in SoC.....	23
4.1 System-on-a-Chip Design Methodology	23
4.2 Reconfigurable Hardware in SoC	24
4.2.1 Using HDLs	24
4.2.2 Using Reconfigurable IP.....	27
4.2.3 Reconfigurable Chips as the SoC	32
4.3 Motivating Reconfigurable Hardware in SoC	33
5 Research Framework	36
5.1 Totem-PLA	36
5.1.1 Circuits.....	36
5.1.2 Delay Model.....	38
5.2 Totem-CPLD.....	43
5.2.1 Circuits.....	43
5.2.2 Delay Model.....	46
5.2.3 Area Model	48
6 Domain-Specific PLAs and PALs	49
6.1 Tool Flow.....	49
6.2 Architecture Generator.....	51

6.3	Layout Generator	57
6.4	Methodology	59
6.5	Results.....	60
6.5.1	Architecture Generator.....	60
6.5.2	Layout Generator	69
6.6	Conclusions.....	69
7	Logic in Domain-Specific CPLDs	72
7.1	Tool Flow	73
7.2	Architecture Generator.....	73
7.2.1	Search Algorithms	78
7.2.2	Algorithm Add-Ons	98
7.3	Layout Generator	101
7.4	Methodology	103
7.4.1	Failed PLAmapping Runs	104
7.5	Results.....	105
7.5.1	Benefits of Domain-Specific Devices.....	111
7.5.2	Using Other Evaluation Metrics	114
7.6	Conclusions.....	115
8	Routing in Domain-Specific CPLDs	116
8.1	Crossbars.....	117
8.2	Sparse-Crossbar Generation.....	119
8.2.1	Initial Switch Placement	122
8.2.2	Moving Switches	125
8.2.3	Algorithm Termination	128
8.2.4	Switch Smoothing.....	129
8.3	Routing for CPLDs with Sparse Crossbars.....	134
8.4	Determining Switch Density of Sparse Crossbars	136
8.5	Results.....	136
8.5.1	Using Other Evaluation Metrics	139

8.6	Conclusions.....	140
9	Adding Capacity to Domain-Specific CPLDs	142
9.1	Adding Capacity to CPLDs	143
9.2	Methodology.....	144
9.2.1	Routing Failures.....	146
9.3	Results.....	148
9.4	Conclusions.....	156
10	Conclusions and Future Work	158
10.1	Contributions.....	158
10.2	Conclusions and Future Work	160
	References.....	168
	Appendix A: Kuhn/Munkres Algorithm.....	173
	Appendix B: Layout Units	178

List of Figures

Figure 1. An SRAM cell.....	6
Figure 2. Wiring using an SRAM controlled transistor.....	7
Figure 3. Using wires and switches to create a full crossbar.....	7
Figure 4. Conceptual diagram of a PLA.....	8
Figure 5. Representing array switches with dots.....	9
Figure 6. Representing gates in a PLA.....	10
Figure 7. PLA representation we will use throughout this work.....	10
Figure 8. A pseudo-nMOS (sense-amplifying) PLA.....	11
Figure 9. Complete Network PLA implementation.....	12
Figure 10. Full, depopulated, and sparse crossbars.....	13
Figure 11. The Xilinx CoolRunner XPLA3 CPLD Architecture.....	14
Figure 12. The Xilinx CoolRunner XPLA3 CPLD Functional Block.....	15
Figure 13. The Xilinx CoolRunner XPLA3 CPLD Macrocell.....	16
Figure 14. CAD flow for programming reconfigurable hardware.....	17
Figure 15. Representing a circuit with a DAG.....	18
Figure 16. Sample circuit after PLAMap's labeling stage.....	20
Figure 17. Sample circuit after PLAMap's mapping stage.....	21
Figure 18. Sample circuit after PLAMap's packing stage.....	22
Figure 19. Directional (left) and gradual (right) Wilton architectures.....	25
Figure 20. Product-term-based synthesizable architectures.....	27
Figure 21. Glacier PLA (GPLA).....	28
Figure 22. A tile in the RaPiD array.....	31
Figure 23. The Totem-RaPiD tool flow.....	32
Figure 24. The RC model for an inverter.....	38
Figure 25. Path propagation through a PLA.....	40
Figure 26. Transforming circuits to BLIF format.....	44

Figure 27. Delay path in a CPLD.....	47
Figure 28. PLA/PAL Generation Tool Flow	50
Figure 29. Mapping circuits in PLAs/PALs	52
Figure 30. Options for using the Kuhn/Munkres algorithm	53
Figure 31. Suboptimality introduced by the Kuhn/Munkres algorithm.....	54
Figure 32. Cost function nuances.....	55
Figure 33. Allowable annealing moves	56
Figure 34. Tiling pre-made layout cells to create PLAs	58
Figure 35. A PLA created using pseudo-nMOS	59
Figure 36. Possible cost reductions using our metric	61
Figure 37. A representative portion of the <i>shift</i> circuit.....	63
Figure 38. Bit reduction vs. percent array utilization	67
Figure 39. Bit reduction vs. connection count agreement	67
Figure 40. Failure of the compactor to reduce area	70
Figure 41. Totem-CPLD Tool Flow	73
Figure 42. CPLD Architecture Generator.....	74
Figure 43. A 1-D search through the PLA space, input step	77
Figure 44. A 1-D search through the PLA space, output step	77
Figure 45. A 1-D search through the PLA space, product term step.....	78
Figure 46. General pseudocode for the search algorithms.....	79
Figure 47. Hill Descent Algorithm	80
Figure 48. Pseudocode for the input step of the Hill Descent algorithm.....	81
Figure 49. Pseudocode for the output step of the Hill Descent algorithm.....	82
Figure 50. Pseudocode for the product-term step of the Hill Descent algorithm	83
Figure 51. Input optimization step of the Successive Refinement algorithm.....	84
Figure 52. Successive refinement trimming	85
Figure 53. Pseudocode for the input step of the Successive Refinement algorithm.....	86
Figure 54. Pseudocode for trimming	87
Figure 55. Pseudocode for the output step of the Successive Refinement algorithm.....	88

Figure 56. Pseudocode for the product-term step of Successive Refinement.....	89
Figure 57. Choose N Regions Algorithm	90
Figure 58. Pseudocode for the input step of the Choose N Regions algorithm.....	91
Figure 59. Pseudocode for the output step of the Choose N Regions algorithm.....	92
Figure 60. Pseudocode for the product-term step of the Choose N Regions algorithm ..	93
Figure 61. Run M Points Algorithm	94
Figure 62. Pseudocode for the input step of the Run M Points algorithm.....	95
Figure 63. Pseudocode for the output step of the Run M Points algorithm.....	96
Figure 64. Pseudocode for the product-term step of the Run M Points algorithm	97
Figure 65. Pseudocode for the radial search add-on	99
Figure 66. Top-level pseudocode when using a second algorithm iteration.....	100
Figure 67. Top-level pseudocode when using the small PLA inflexibility add-on	101
Figure 68. CPLD floorplan and layout	102
Figure 69. Determination of N in Choose N Regions algorithm	106
Figure 70. Determination of M in Run M Points algorithm	107
Figure 71. Our use of full and sparse crossbars	117
Figure 72. Full crossbar connectivity.....	117
Figure 73. Minimal full crossbar	118
Figure 74. Maximizing connectivity in crossbars.....	120
Figure 75. Representing crossbar input lines as vectors of 1s and 0s.....	120
Figure 76. Crossbars, their vector representations, and their hamming distances	121
Figure 77. Top level pseudocode for the switch placement algorithm	122
Figure 78. Pseudocode for initial switch placement algorithm.....	123
Figure 79. Sample initial switch placement.....	124
Figure 80. Allowable switch moves.....	126
Figure 81. Pseudocode for the switch movement algorithm.....	127
Figure 82. Routability results from several different switch patterns.....	128
Figure 83. Routing full crossbars.....	130
Figure 84. Routing sparse crossbars	130

Figure 85. Ideal switch placement for layout.....	131
Figure 86. Pseudocode for the switch smoothing algorithm.....	132
Figure 87. Switch smoothing example.....	133
Figure 88. Representative layouts of unsmoothed and smoothed crossbars.....	133
Figure 89. A simple CPLD and its routing graph	135
Figure 90. Binary Search pseudocode	137
Figure 91. Evaluating CPLD augmentation strategies.....	145
Figure 92. Pseudocode for spreading out CPLD mappings.....	147
Figure 93. Results of adding switches to our crossbars	150
Figure 94. Results of adding specific logic resources.....	151
Figure 95. Results of adding specific logic resources, best case	153
Figure 96. Results of augmenting PLA size	154
Figure 97. Results of using hybrid augmentation strategy	156
Figure 98. Difficulties of automatic 2-D FPGA generation	165
Figure 99. Setting up the Kuhn/Munkres algorithm	173
Figure 100. Calculating product term costs for Kuhn/Munkres	174
Figure 101. Turning our product term problem into maximal perfect matching.....	174
Figure 102. Initial valuse for f in Kuhn/Munkres	175
Figure 103. The graph G , the values of f , and the spanning subgraph G_f	175
Figure 104. Pseudocode for the Kuhn/Munkres algorithm.....	176
Figure 105. Units in our CPLDs	180
Figure 106. Programmable switch from the CPLD crossbar	181
Figure 107. Inverter and buffer that feed the PLA's AND-plane.....	182
Figure 108. AND-plane connection.....	183
Figure 109. Pullup transistor for the AND-plane.....	184
Figure 110. Buffer between the AND-plane and OR-plane	185
Figure 111. OR-plane connection	186
Figure 112. Pullup transistor for the OR-plane.....	187
Figure 113. Inverter appearing after the OR-plane.....	188

Figure 114. D-Flip-Flop used for optional registering of the PLA outputs 189
Figure 115. 2-to-1 multiplexor for choosing the registered/unregistered PLA output .. 190
Figure 116. SRAM bit used for controlling the 2-to-1 multiplexor..... 191

List of Tables

Table 1. The circuits used for Totem-PLA	37
Table 2. The sample domains used for preliminary testing	44
Table 3. The main domains used in our work.....	45
Table 4. Additional domains used for Chapter 9 results.....	46
Table 5. The circuits, with their information and groupings	60
Table 6. Running the PLA/PAL algorithms on multiple occurrences of acircuit.....	62
Table 7. Results of PLA-Fixed and PLA-Variable algorithms vs circuit count	65
Table 8. Reductions in programmable bits and delay for PLA/PAL algorithms.....	68
Table 9. Results for different PLA parameters in our test circuits	75
Table 10. Results for different PLA parameters in our test circuits, refined.....	76
Table 11. Architecture results for our domain-specific algorithms	107
Table 12. Base algorithm results compared to using a second iteration.....	108
Table 13. Search algorithms run with a second iteration.....	109
Table 14. Search algorithms run with a second iteration, using old models	110
Table 15. Domain-specific architecture performance vs fixed architectures	111
Table 16. Mapping results for the three largest circuits in the floating-point domain....	112
Table 17. Mapping results for the four largest circuits in the arithmetic domain.....	112
Table 18. Running each domain on the best domain-specific architectures.....	113
Table 19. Running each domain on the consensus architecture	113
Table 20. Results for area-delay driven, area driven, and delay driven modes	114
Table 21. Area of routing resources in CPLDs.....	116
Table 22. Switch smoothing algorithm results	134
Table 23. Full-crossbar results vs sparse-crossbar results	138
Table 24. Switch densities of sparse crossbars	138
Table 25. Best results found for full and sparse-crossbar-based CPLDs.....	138
Table 26. Results of area-delay driven, area driven, and delay driven modes	139

Table 27. Strategies for adding capacity to our CPLD architectures.....	144
Table 28. Results of adding switches to our crossbars	150
Table 29. Results of adding specific logic resources.....	151
Table 30. Results of adding specific logic resources, best case.....	152
Table 31. Results of augmenting the PLA size.....	153
Table 32. Results of using hybrid augmentation strategy.....	155

1 Introduction

As the semiconductor industry continues to follow Moore's Law, a switch in design paradigm is occurring. The former "System-on-a-Board" style, which had several discrete components individually fabricated and then integrated together on a board, is becoming obsolete. Because of the constant increase in gate count (currently chips can hold hundreds of millions of wirable gates), we are at a point where distinct VLSI components can now be incorporated onto a single silicon chip. This "System-on-a-Chip" (SoC) methodology is becoming very popular, as its benefits include improved area/delay/power characteristics as well as increased inter-device communication bandwidth.

A drawback of the SoC methodology is that chip designs are much larger and more complicated than in previous design methods. Designs tend to be more involved and time consuming due to the need to tightly integrate multiple components onto a single substrate. A result of this is that design decisions tend to be cemented earlier in the design process, as any design modification will likely affect the integration of the entire system. This leaves little room for any sort of modifiability late in the design cycle, regardless of how useful such modifications might be.

An elegant solution to this problem is to include reconfigurable logic on SoC devices. Reconfigurable logic is composed of a mixture of routing and logic resources that are controlled by SRAM bits, such that programming the SRAM bits allows designers to implement designs directly in hardware. Designing in hardware provides the benefits of high speed and low power, while the reprogrammability of these devices gives them much of the flexibility of a general-purpose processor. Including reconfigurable logic on an SoC would allow designers to make design decisions late in the design cycle, as they could implement these aspects on the reconfigurable fabric. Additionally, the

reconfigurable logic can be leveraged for general computing ability, can provide run-time reconfigurability, or could even be used as a means for providing upgradeability for a device that is already being used in its target environment.

Traditional reconfigurable logic devices can provide good performance, but they do not approach the performance provided by an application specific integrated circuit (ASIC). In order to retain generality, and to be able to support a large number of disparate designs, reconfigurable devices need to have a large amount of flexibility. The programming overhead required by this flexibility, however, is exactly what prevents reconfigurable devices from performing as well as ASICs. As such, it would be useful to remove unnecessary flexibility from a reconfigurable device in order to reduce the area, delay, and power penalties that are incurred.

One way to remove flexibility from a reconfigurable device is to tailor it to a particular domain. A “domain-specific” reconfigurable architecture is one that is designed to efficiently support a specific application domain, but which does not need to support any sort of design outside of this domain. By specifying the applications that the reconfigurable logic must support, the flexibility of the architecture can be greatly reduced, resulting in greater performance.

Many examples of domain-specific reconfigurable architectures exist. Within academia, RaPiD [1, 2] and Pleiades [3] have been created to target the DSP domain, while PipeRench [4] has been created for use within the multimedia domain. Domain-specific commercial architectures have also started to appear, as Xilinx’s Virtex-4 devices come in three different flavors: “Ultra-high-performance signal processing”, “Embedded processing and high-speed serial connectivity”, and “High-performance logic” [5].

All of these domain-specific architectures took a great amount of time and effort to design, however. An SoC designer might not be willing to wait several months for a domain-specific reconfigurable architecture to be created for their chip, as it would likely delay the completion of the device. The dilemma thus becomes creating these domain-

specific reconfigurable fabrics in a short enough time that they can be useful to SoC designers.

The Totem project is our attempt to reduce the amount of effort and time that goes into the process of designing domain-specific reconfigurable logic. By automating the generation process, we will be able to accept a domain description and quickly return a reconfigurable architecture that targets the specific application domain, allowing SoC designers to get an efficient architecture with no adverse effects on their design schedule. Early work in Totem [6-25] has leveraged RaPiD, a one-dimensional word-wide architecture that uses coarse-grained units (ALUs, Multipliers, RAMs, etc.) to target applications within the DSP domain.

This work deals with the creation of domain-specific CPLD architectures for use in SoC, a project termed Totem-CPLD. CPLDs are relatively small reconfigurable architectures that typically use PLAs or PALs as their functional units, and which connect the units using a single, central interconnect structure. In commercial architectures, the functional units tend to be relatively coarse-grained in order to provide shallow mappings, leading to low, predictable delays.

- *Chapter 2: Programmable Logic Devices* introduces the concept of reconfigurable hardware, providing technical background for the reconfigurable devices that are applicable to the work described in this dissertation.
- *Chapter 3: CAD for Programmable Logic* introduces the basic CAD design flow for reconfigurable devices and provides details for some of the algorithms that we will be using.
- *Chapter 4: Reconfigurable Hardware in SoC* provides an introduction to the System-on-a-Chip methodology, and provides examples of reconfigurable architectures that have been created with the SoC device in mind.
- *Chapter 5: Research Framework* provides the experimental foundation of the work presented here, including information on the benchmark sets used and the methods employed in evaluating our results.

- *Chapter 6: Domain-Specific PLAs and PALs* takes a look at the logic units typically used in CPLDs and examines the ways in which they can be tailored to an application domain.
- *Chapter 7: Logic in Domain-Specific CPLDs* introduces the algorithms used to tailor the logic in a full-crossbar-based CPLD to a specific application domain.
- *Chapter 8: Routing in Domain-Specific CPLDs* introduces the use of sparse switch matrices in the CPLD crossbar structures, and discusses how to tailor these routing structures to a specific application domain.
- *Chapter 9: Adding Capacity to Domain-Specific CPLDs* provides an analysis of which CPLD characteristics are most essential for providing support for future untested designs, ensuring that they will fit onto the CPLD architecture.
- *Chapter 10: Conclusions and Future Work* provides a summary of the contributions of this work, as well as suggestions for future work that can be performed.

2 Programmable Logic Devices

Programmable logic devices (PLDs) are a popular solution for systems where fast turn-around-time and/or low cost are high priorities. PLDs achieve these features through post-fabrication modifiability, as they are capable of changing the circuits that they implement in response to user-supplied specifications. This chapter provides the technical background necessary for understanding PLD architectures, with particular emphasis on the Complex Programmable Logic Device (CPLD), as this is the primary architecture that we consider in this dissertation. In order to familiarize the reader with typical device specifications, we also provide a close examination of a popular commercial CPLD, the Xilinx CoolRunner XPLA3.

2.1 Programmable Logic Devices

PLDs are devices that are capable of being dynamically reconfigured at any time to implement new designs. They achieve this by containing routing and logic resources that are controlled by modifiable memories: adjust the memory contents and you adjust the routing structure and/or the functions being implemented by the logic resources.

Different styles of memory have been employed in the creation of PLDs, from basic EEPROM (electrically erasable programmable read-only memory), to flash memory, to SRAM (static random access memory). EEPROM and flash seem to be the preferred memory style for CPLDs [26-29], as they have the attractive characteristic of retaining their contents when powered off, allowing instant logic availability at boot-up. EEPROM and flash memories retain their contents because they use floating-gate transistors: application of extra high or low voltage differentials across a floating gate causes tunneling to deposit or remove electrons from the floating gate, changing the threshold of the EEPROM or flash transistor. Regular operating voltages have negligible effects upon

these floating gates, just as the absence of power does not effect them, so they are able to retain their programmed states even when powered off.

One constraint on our designs is that we must implement our reconfigurable architecture in the same process technology that the rest of the SoC is going to be using, since everything will be implemented on a single piece of silicon. While floating gates are useful in creating certain memories, they are not particularly useful in the design of processors, DSPs, or custom logic, and it is these other blocks that will dictate the process technology of the SoC. EEPROM and flash memories are therefore not reasonable memory choices for us, because the SoC fabrication process won't have the ability to create the necessary floating gates.

SRAM memory cells, on the other hand, are created without the need for floating gates. They can be fabricated using the same technology that most SoC devices will use, and are therefore an attractive memory solution for our reconfigurable architectures. We will be using SRAM as the configuration memory in our work, and it will be laid out as individual SRAM cells as shown in Figure 1.

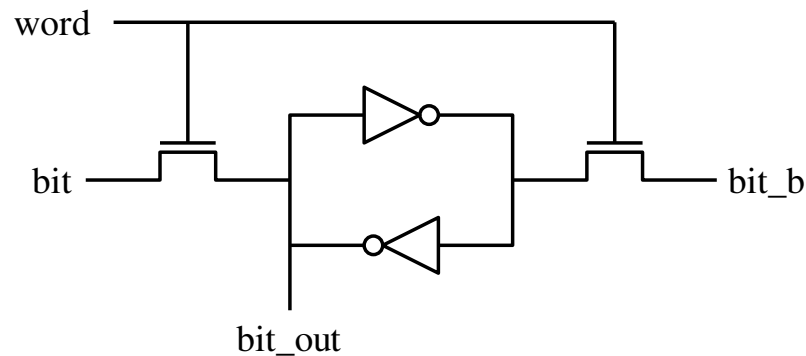


Figure 1. A simple 6-transistor SRAM cell. The cell is programmed by setting word=1 and applying the proper values to bit and bit_b. bit_out represents the value of the cell

Figure 2 shows how a single SRAM cell can be used to configure routing resources. In the figure, two wires are connected by a transistor, with the gate of the transistor being controlled by an SRAM cell (the circled P will be used to denote a single SRAM bit from here forward). By setting the bit to 1, signals can be driven across the transistor. By setting the bit to 0, the horizontal and vertical wires are isolated.

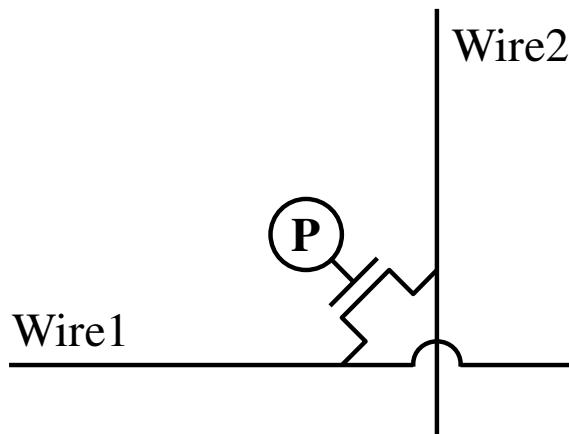


Figure 2. An SRAM controlled transistor can isolate or connect wire1 and wire2

Wires and configurable switches like the one shown in Figure 2 can easily be used to create a full crossbar, as shown in Figure 3. We now have several horizontal wires that can each be connected to any of several vertical wires. In the normal operation of a crossbar, each output wire is connected to exactly one input wire, with the numbers of output wires less than or equal to the number of input wires. Figure 3 also shows the simplified drawing of this crossbar, representing the configurable switches with dots. This is how we will draw the crossbar in future diagrams.

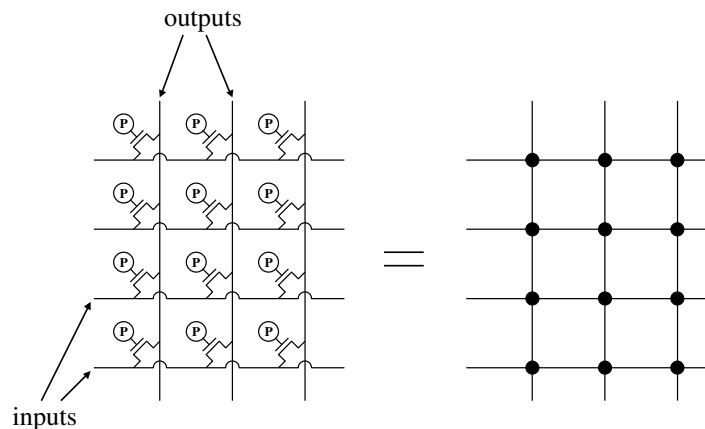


Figure 3. Using wires and switches to create a full crossbar. Right, the configurable switches are represented by dots

One method of performing logic in PLDs is through the use of a programmable logic array, or PLA. PLAs are devices that directly implement two level sum-of-products style logic functions. They do this with the use of a programmable AND-plane that leads to a

programmable OR-plane, as shown in Figure 4. Input signals come into the array in both true and negated form, and the appropriate signals are fed as inputs to the AND gates as determined by configurable switches. The outputs of these AND gates are then selectively fed to OR gates, again controlled by configurable switches. The outputs of the OR gates can then be registered or used as combinational signals, as determined by the bit controlling the output multiplexer. Note that the actual hardware implementation of a PLA, discussed later in this chapter, is actually much more efficient than what is shown in the figure.

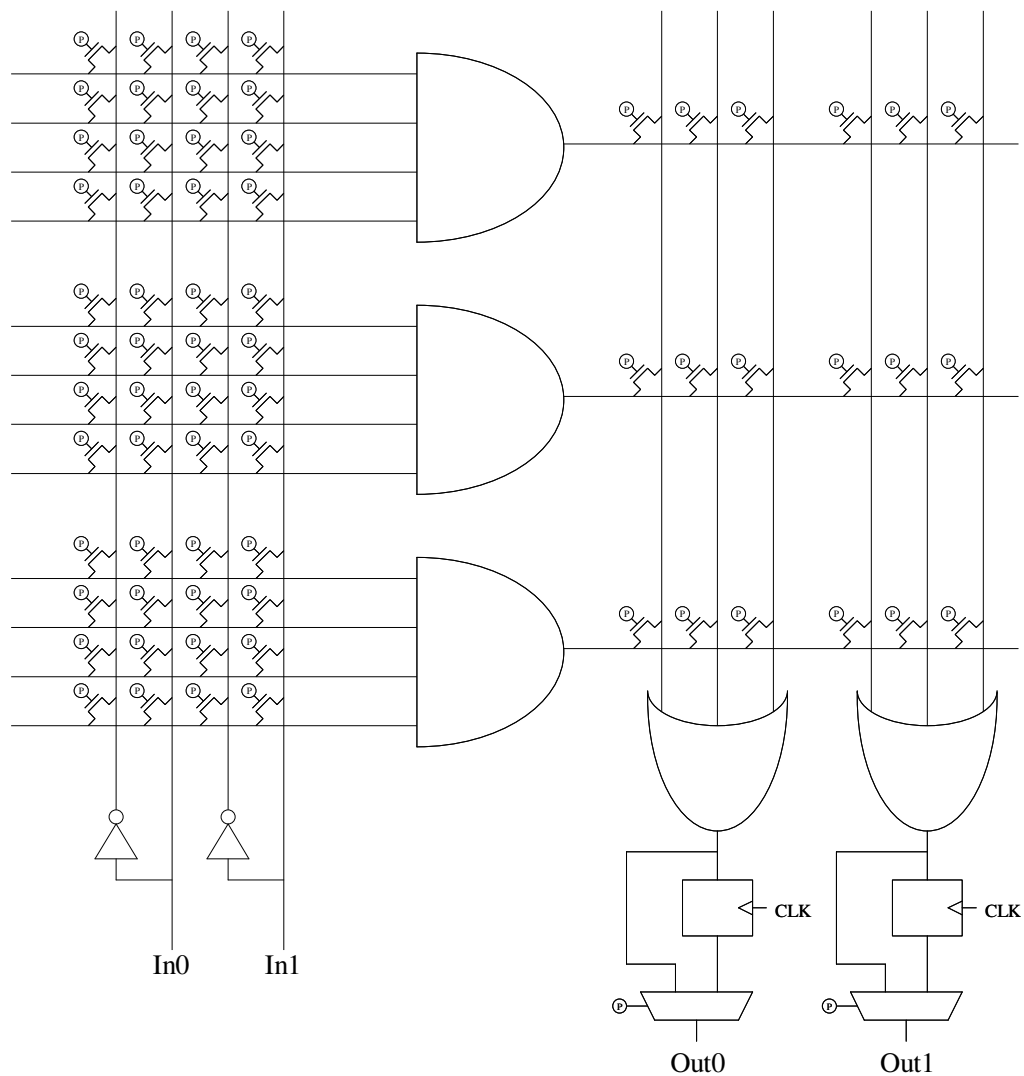


Figure 4. Conceptual diagram of a PLA, with a programmable AND-plane (left) and a programmable OR-plane (right), including optional registering of the outputs

When drawing a PLA, it is common to omit the switches that connect orthogonal wires in the arrays and to simply put dots in locations where the switches are configured to connect wires. Figure 5 displays this transition, as the left half shows a configuration with the programmable bits set to 1 (conducting) or 0 (not conducting), and the right half shows the same function in the simplified representation. In future diagrams we will also not be drawing the optional registering at the output of the PLA, despite the fact that it always exists. This is done to make the diagram less cluttered.

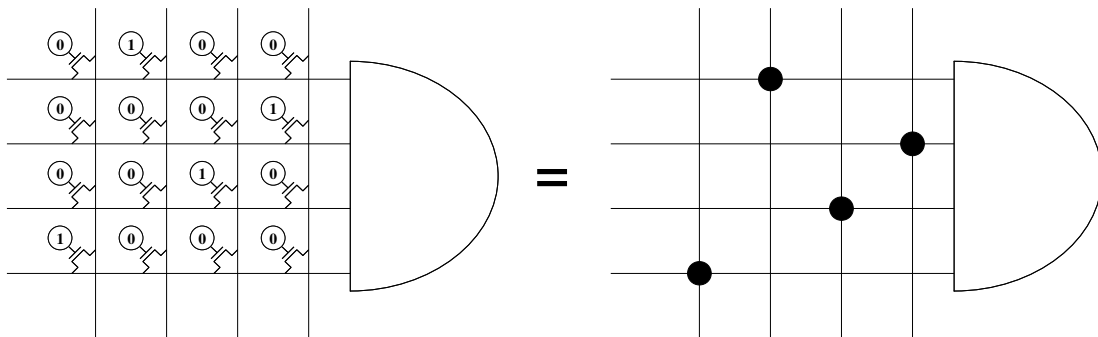


Figure 5. Switches that are turned on in PLA arrays will be represented by dots. This is shown for the AND-plane, and is also true for the OR-plane

Another convention is to omit the AND and OR gates in the drawing, and to merge their respective input wires into one wire. Using this representation, the horizontal wires that leave the AND-plane represent the outputs of AND gates, and the vertical wires that leave the OR-plane represent the outputs of OR gates. Figure 6 displays this transition, as the left half depicts the individual wires that feed the gates, and the right half shows the same function being implemented in simplified form.

To give a specific example of this new simplified representation, Figure 7 displays a small PLA. The functions being implemented by the array are also listed in the figure. Notice that PLA arrays are defined by how many inputs, product terms, and outputs they can represent. The PLA in the figure has four inputs, three product terms, and two outputs, which will be written in shorthand as a 4-3-2 PLA.

Most PLAs are not actually implemented with AND and OR-planes, but instead with two NOR-planes. As Equation 1 shows, equations in sum-of-products (or AND-OR) form can easily be represented in NOR-NOR form by applying De Morgan's law. In

order to ensure that the same equation is being implemented, we must simply negate all the input values and then negate the subsequent output. Since we already have true and negated forms of each input, this will be easy to do.

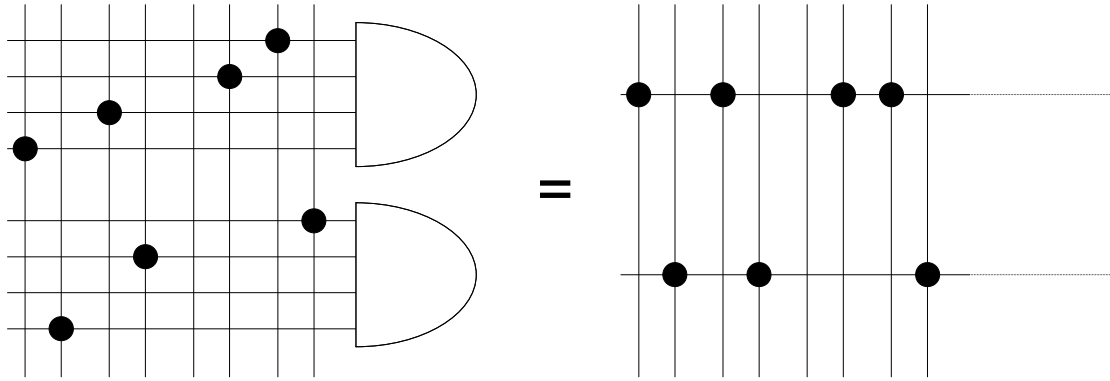


Figure 6. The gates will be represented by individual wires, and all inputs to the gates will be connected by dots to the line. This is shown for the AND gates, and is also true for the OR gates

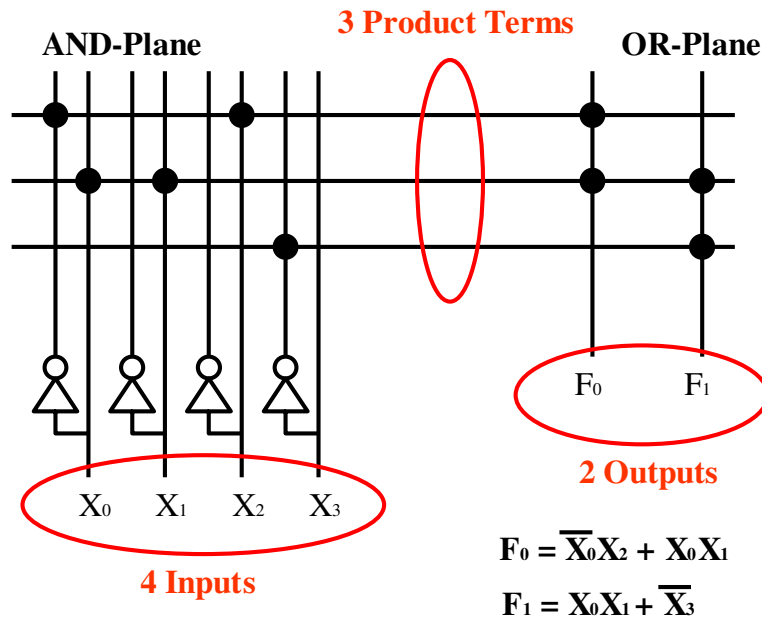


Figure 7. The representation of a PLA that we will use throughout this work. PLAs are specified by the number of inputs, product terms, and outputs that they can represent

$$AB + C\overline{D} + \overline{A}D = \overline{\overline{A} + \overline{B} + \overline{C} + D} + \overline{\overline{A} + D} = \overline{\overline{A} + \overline{B} + \overline{C} + D + A + D} \tag{1}$$

The method in which configurability is implemented in a PLA depends upon the implementation style of the array, but NOR-NOR PLAs are particularly well suited to

pseudo-nMOS (also called sense amplifying) implementations, as shown in Figure 8. Using this style, the array connections need only consist of two small series pull-down transistors: the first transistor's gate is controlled by the input line, and the second transistor is controlled by a user-specified SRAM bit, controlling whether the output line can actually be discharged by making the first transistor conduct. An additional advantage to this style is that only pull-up transistors are needed at the edges of the arrays. It can be seen that each plane implements the NOR function, as plane-outputs are initially charged to high through the pull-up transistors, and any array input that is a 1 will pull the output to low. Pseudo-nMOS PLAs have very compact layouts and very reasonable delay characteristics, so this is the implementation style that we use.

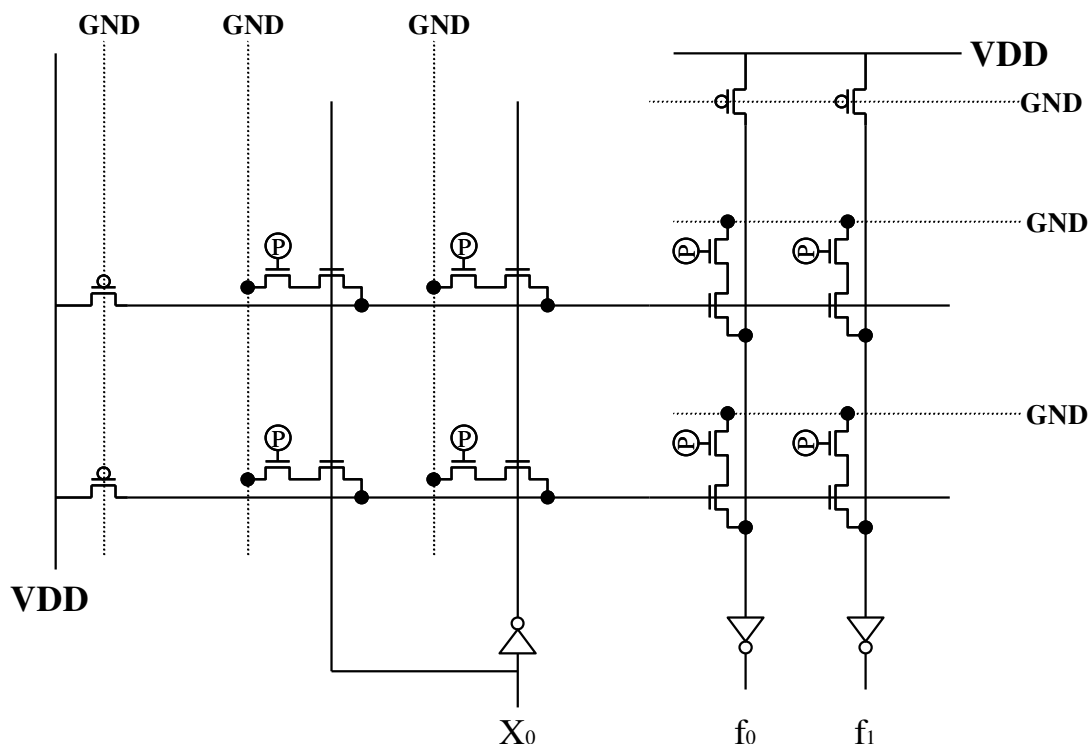


Figure 8. A PLA implemented in pseudo-nMOS (also called "sense-amplifying") style

There is another type of array that directly performs sum-of-products style logic, and it is called a PAL. A PAL differs from a PLA in only one way: rather than having a fully programmable OR-plane, a PAL has fixed OR gates. Because of this, PALs tend to be

slightly smaller than PLAs, but they are not as flexible as PLAs due to the fixed nature of their OR implementation.

2.1.1 Complex Programmable Logic Devices (CPLDs)

All of the components that we've just introduced can be combined to form a popular style of PLD, the Complex Programmable Logic Device (CPLD). CPLDs use either PLAs or PALs as their functional units, and typically connect the functional units together using a crossbar. Because they use crossbars as their interconnect structures, and crossbars grow quickly in size, CPLDs have historically been limited to implementing small to medium sized designs.

Routing in a CPLD is typically done through the use of a "complete network". Using this method, every device input (from I/O) and every PLA/PAL output directly drives a wire that traverses the length of the device, as shown in Figure 9. Connections to the PLA/PAL inputs are then made using crossbars. If full crossbars are used, then the problem of routing signals in a CPLD is trivial because the crossbar delivers "full capacity".

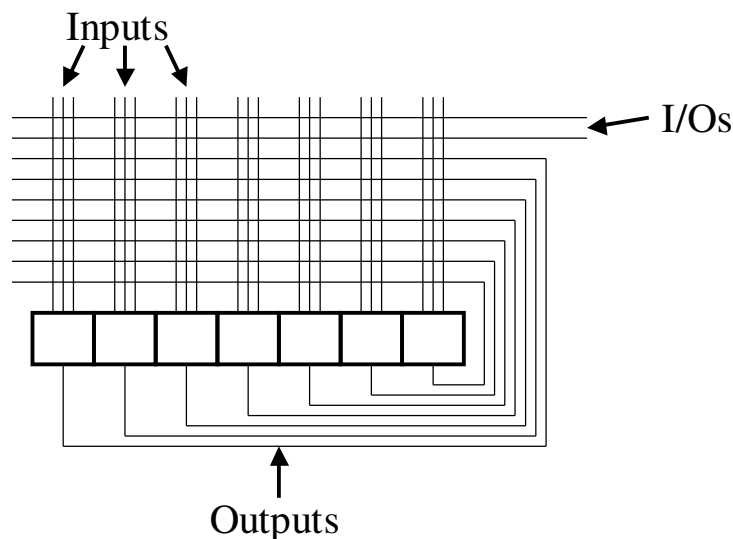


Figure 9. A "Complete Network" in which all outputs and I/Os are available as inputs to all units

A crossbar delivers full capacity if any subset of inputs wires (obeying $numInputsInSubset \leq numOutputs$) can reach output wires. If depopulated crossbars are

used, such that not all of the crosspoints in the crossbars have switches, then routing algorithms will need to be employed if full capacity is not still guaranteed. We will call crossbars that do not provide full capacity “sparse” crossbars. Figure 10 displays a full crossbar, a depopulated crossbar that still provides full capacity (choose any set of four or fewer inputs and they can be connected to outputs), and a sparse crossbar that does not provide full connectivity.

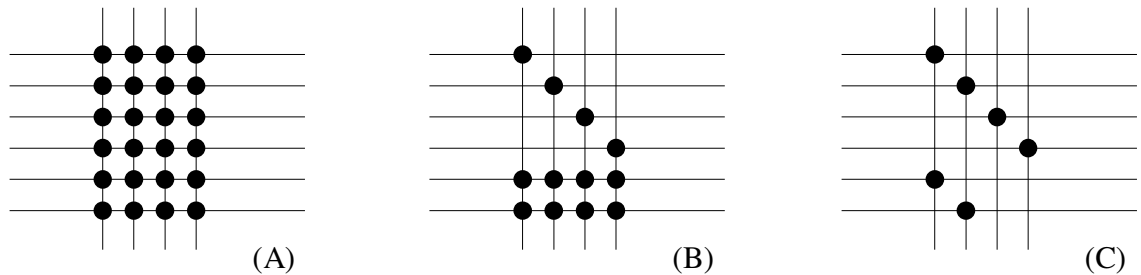
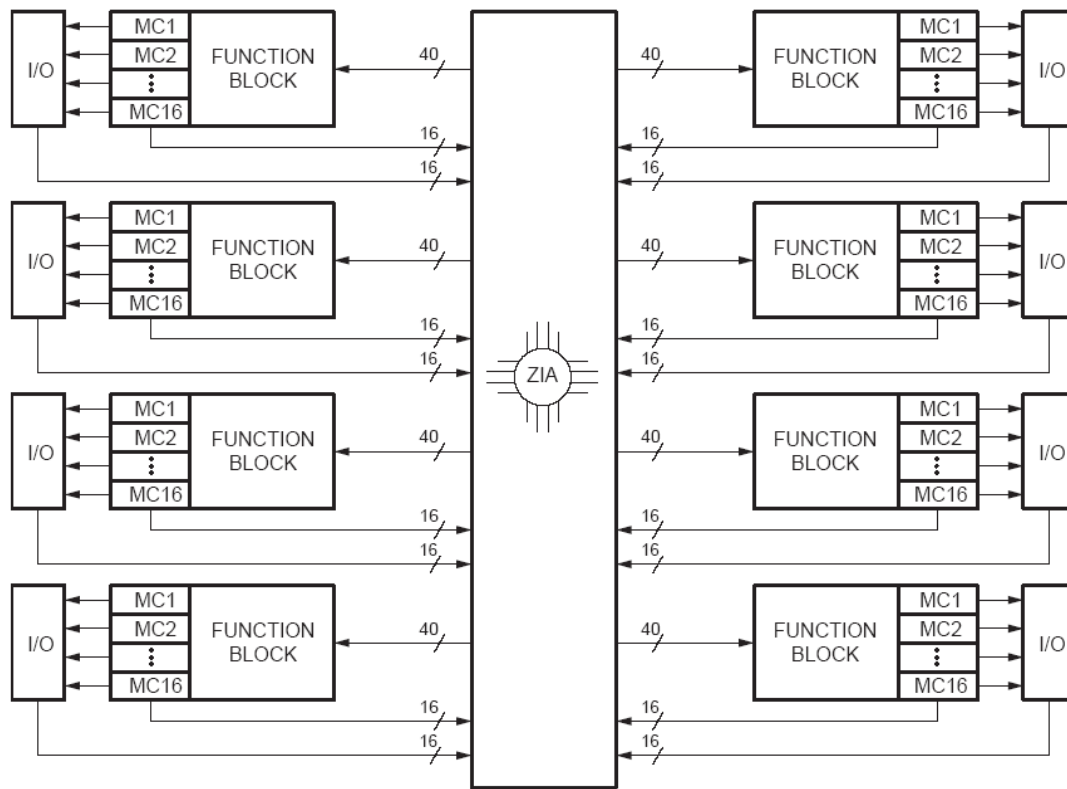


Figure 10. A full crossbar (A), a depopulated crossbar that still provides full capacity (B), and a sparse crossbar that does not provide full capacity (C)

2.1.2 Case Study – Xilinx CoolRunner XPLA3 CPLD

The CPLD description given above is very basic, and commercial devices tend to add many bells and whistles to the basic CPLD in order to increase performance and flexibility. In order to further familiarize the reader with CPLD architectures, this section gives a discussion of the Xilinx CoolRunner XPLA3 CPLD [26].

Figure 11 shows the Xilinx XPLA3 CPLD Architecture. A Function Block and 16 Macrocells (MC) combine to form each PLA, and the Zero-power Interconnect Array (ZIA) is a virtual crosspoint switch, providing full connectivity between the PLAs. All I/Os and PLA outputs feed the ZIA, and PLA inputs are pulled from the ZIA. The basic PLA has 40 inputs, 48 product terms, and 16 outputs. These devices come in varying sizes, ranging from 32 Macrocells (2 PLAs) to 512 Macrocells (32 PLAs). A 128 Macrocell device is shown. I/O capacities range from 36 I/Os for a 32 Macrocell device, to 260 I/Os for a 512 Macrocell device.



DS012_01_112000

Figure 11. The Xilinx CoolRunner XPLA3 CPLD Architecture [26]

Figure 12 shows the functional block in more detail. The basic PLA can be seen in this diagram, as 40 inputs from the ZIA feed the Product-Term Array, creating 48 product terms that get ORed (along the bottom) to create 16 different outputs, with one output going to each Macrocell. Xilinx has also added functionality to this basic PLA. Wider logic equations can be synthesized using the eight Foldback NAND product terms at the top of the diagram. Eight other product terms (PT0 – PT7) can be used as control signals for the register that exists in the Macrocell. Additionally, 16 product terms (PT32-PT47) are available for timing critical signals, and can be fed directly to the Macrocell rather than feeding the ORed product terms to the Macrocell. The Variable Function Multiplexers (VFM), which choose between the high-speed wires and the ORed product term outputs, can also increase logic density by implementing some two input logic functions.

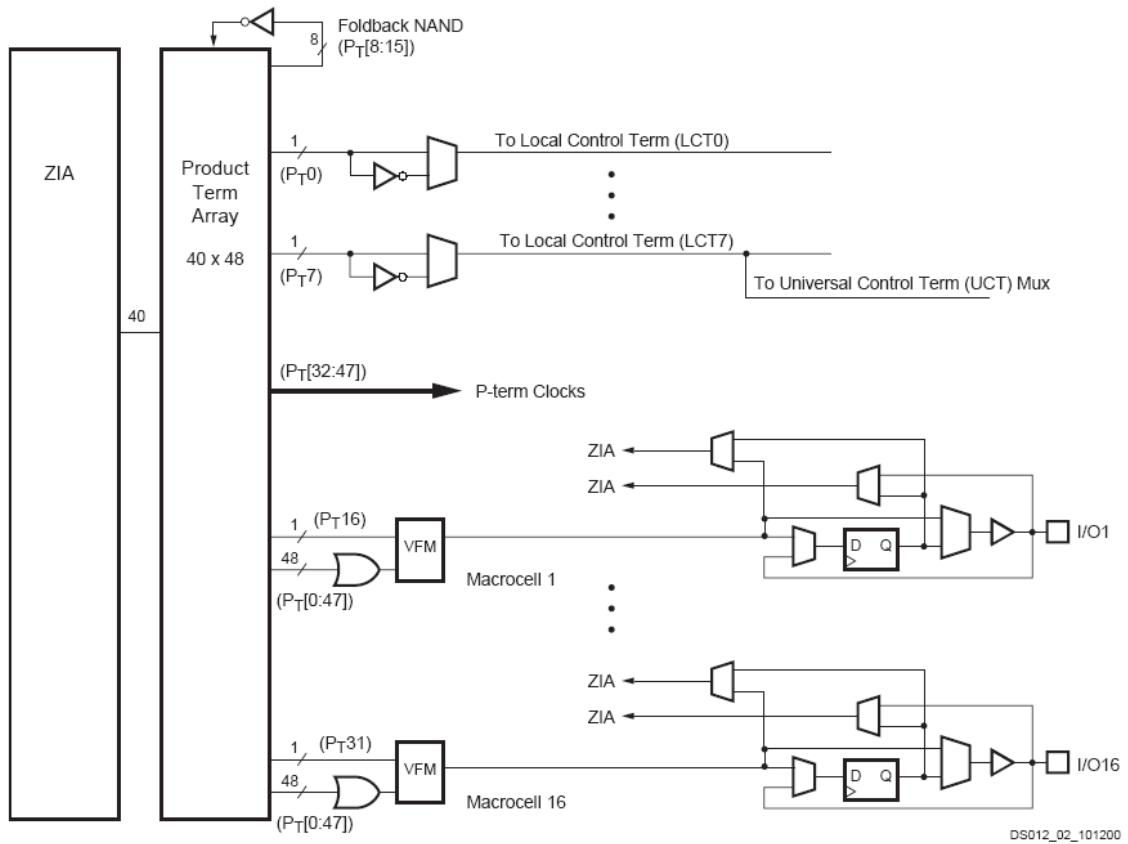
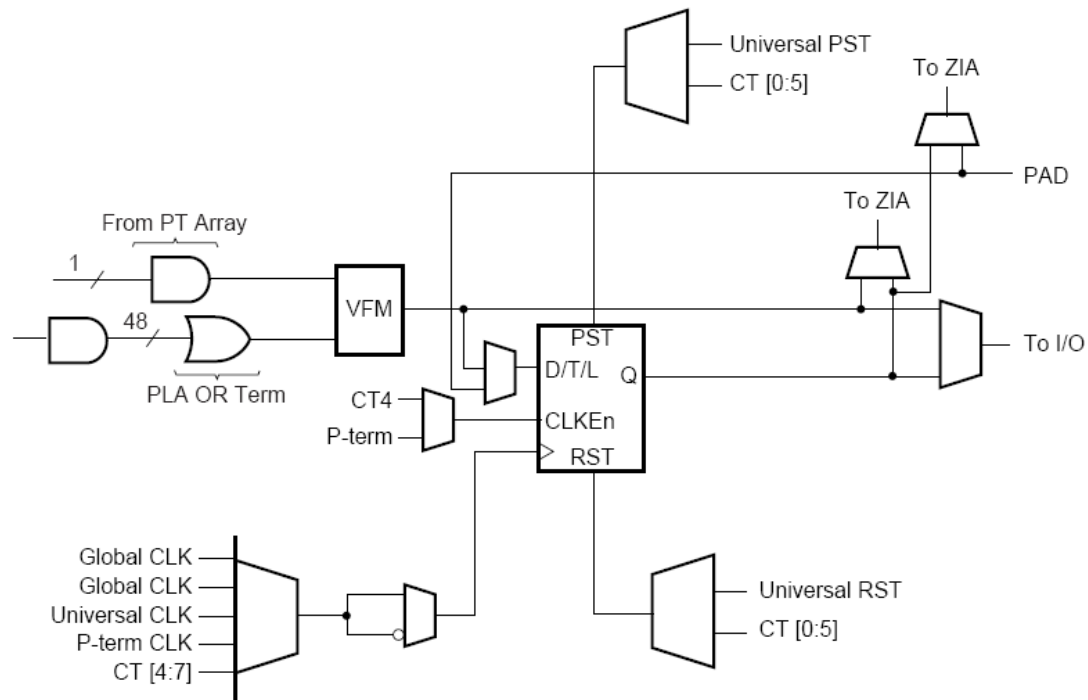


Figure 12. The Xilinx CoolRunner XPLA3 CPLD Functional Block [26]

The Macrocell is shown in Figure 13. The output of the VFM is fed to a register that can be configured as a D-, T-, or Latch-type flip-flop, and which can be controlled by many sources. There are then two muxed paths leading to the ZIA. The first mux selects between the combinational or registered output of the VFM. The second mux selects between the output of the register or the I/O pad of the Macrocell. If the I/O pad is being used as an input, this is how the signal is driven onto the ZIA.



Note: Global CLK signals come from pins.

ds012_05_122299

Figure 13. The Xilinx CoolRunner XPLA3 CPLD Macrocell [26]

The Xilinx CoolRunner XPLA3 CPLD provides all of the basic CPLD functionality described earlier, and also provides hardware that increases logic density and allows for high-speed signal paths. We will not be concerned with creating these optimizations in our own CPLD architectures, but will instead be attempting to tailor the basic PLA/PAL devices in order to optimize them to application domains.

This introduction to programmable logic devices provides sufficient background for understanding the architectural issues present throughout the remainder of this dissertation. If desired, additional details can be obtained by closer examination of the device introduced in this section [26].

3 CAD for Programmable Logic

In the previous chapter we introduced some reconfigurable architectures, and showed how they can be made to represent different designs by configuring the SRAM bits in the architecture. The task of figuring out exactly how to program each SRAM bit is done by Computer-Aided-Design (CAD) tools. A typical CAD design flow is shown in Figure 14. The designer describes their design either in a high-level hardware description language (HDL) or in schematic form, and feeds this to the CAD tools. The design is first synthesized into a gate-level description (if necessary), and then technology mapped in order to use the actual functional units that the PLD architecture contains. The CAD tools then place and route the circuit onto the target device architecture, and output the configuration bitstream that is used to program the actual SRAM bits that reside on the device. Feedback in the design cycle can occur if constraints (i.e. timing or area) are not met.

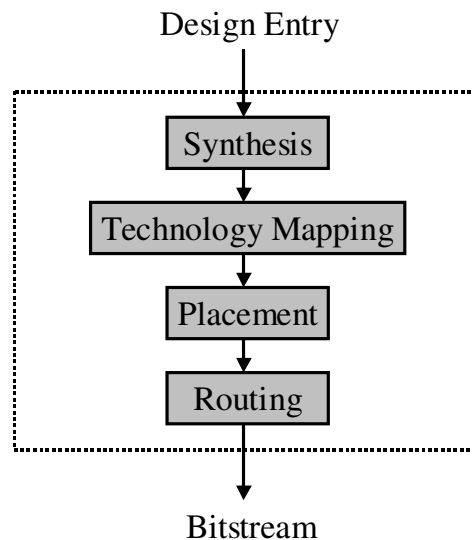


Figure 14. Typical CAD flow for programming reconfigurable hardware

This dissertation is aimed at creating domain-specific CPLDs, so we need to examine how this affects the basic CAD tool flow shown in Figure 14. Our CPLDs will use complete networks, so all of the signals will traverse the length of the architecture by default. This removes our need for a placement algorithm. We will initially use full crossbars to connect the routing and logic resources, so this removes the need for a routing algorithm in our early work. In Chapter 8 we will create architectures that do not use full crossbars, but we will use well-established techniques for the task of routing these architectures. Synthesis for our CPLDs will also be done using standard techniques. This leaves us with the task of technology mapping, which will actually have a direct affect on our methodology and results.

In general, the technology-mapping problem for CPLDs has been most successfully handled by graph-based algorithms, which are created from simple Boolean networks. A Boolean network can be represented as a directed acyclic graph (DAG) where each node is a logic gate and a directed edge (j, k) exists if the output of gate j is an input of gate k . A primary input (PI) node has no input edges and a primary output (PO) node has no output edges. Figure 15 shows the DAG representation of a simple circuit.

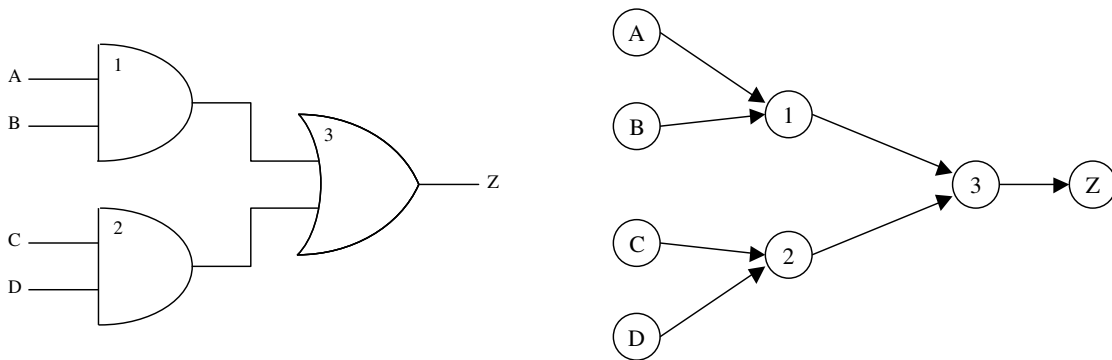


Figure 15. A simple circuit and its DAG representation

Currently, the best technology-mapping algorithm for CPLDs is a tool called PLAmapping [30] which was developed at UCLA in 2001. PLAmapping is a performance driven mapping algorithm whose primary goal is to minimize the delay/depth of the mapped circuit. Heuristics are applied to reduce the area of a mapping, but area optimization is

secondary to delay optimization in this algorithm. This is the technology-mapper that we will employ in this work.

Some background terminology is required to understand the basic PLAMap algorithm. A (k, m, p) -PLA implies a PLA with k inputs, m product terms, and p outputs. A cluster is a subgraph of the network graph with the property that any path connecting two arbitrary nodes in the cluster lies entirely within the cluster. So if PLAMap is mapping to an architecture with (k, m, p) -PLAs, then the goal of the algorithm is to cover the entire Boolean network with (k, m, p) -feasible clusters, which are then converted into PLAs. The clusters need not be disjoint, as nodes can be duplicated as long as the final network is equivalent to the original. The main objective of the algorithm is to minimize circuit delay, which occurs by minimizing the depth of the mapping. This is an NP-hard problem, so PLAMap uses heuristics to tackle it.

The input to PLAMap is a 2-bounded circuit, meaning that it consists of gates with no more than two inputs. A pre-processing step transforms this gate representation into a DAG. The main PLAMap algorithm then works in three steps. In the “labeling” step, each node is given a level (corresponding to circuit depth) that provides clustering information for future steps. Next is the “mapping” step, in which nodes are mapped into specific (k, m, p) -PLAs. Last is a “packing” step, in which PLAMap tries to pack PLAs into each other in order to reduce area.

In the labeling phase, we start by labeling each PI as 0. The nodes are then considered in topological order. For a node v , let l be the maximum label of all its fanin nodes. If node v grouped with the predecessors of v that have label l form a $(k, m, 1)$ -feasible cluster, then we give v the label l . Otherwise we label v with $(l + 1)$. This label represents the logic depth of all the nodes in the network, and the maximum label from this phase will be the depth of the final mapping. Figure 16 gives an example of the labels applied to each node in a sample network that is mapping to 3-3-2 PLAs.

In the second stage, termed “mapping”, the algorithm transforms the $(k, m, 1)$ -feasible clusters into (k, m, p) -feasible clusters based on the labeling information from stage 1. The nodes are considered individually, working backwards from POs to PIs (in

label-decreasing, slack-increasing order, and reordered before each node is considered). If the node currently under consideration has not been put into a cluster yet, then a single-output cluster is formed to include the node and all of its predecessors that have the same label. If this cluster attempts to cover a node that has already been mapped into some other cluster, then one of three things can occur. First, the algorithm attempts to merge the clusters that share the shared node. If this doesn't work, the algorithm attempts to form a reduced cluster that does not include the shared node. This requires "slack" in the system, and if the depth of the mapping is affected by forming this reduced cluster, then the reduced cluster is rejected. This leads to the final (and worst) case, in which the node is simply duplicated.

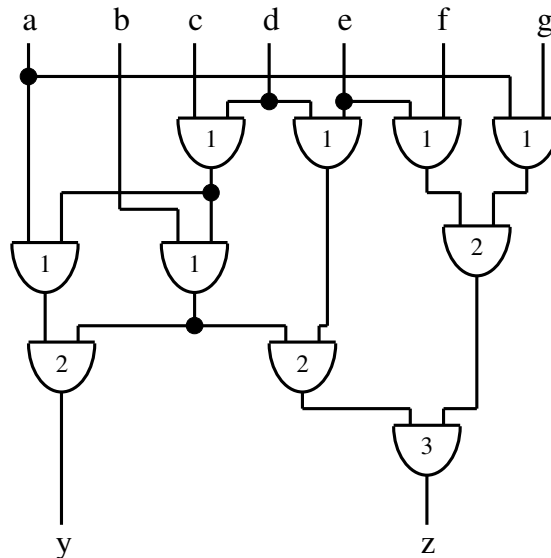


Figure 16. Sample circuit after PLAMap's labeling stage. The labels are listed in the gates

If a node considered in the mapping stage has already been put into a cluster, a problem may arise if the output of this node feeds out of the cluster it is in. First, the algorithm attempts to introduce this output as a new output to the cluster. If it cannot introduce a new output to the cluster, then the node must be duplicated, and the duplicated node forms its own cluster. Figure 17 shows the circuit from Figure 16 after mapping has occurred.

Last, the packing stage occurs, in which the algorithm attempts to reduce the total number of PLAs. The first operation it performs is PLA collapsing, where the algorithm attempts to collapse a PLA into all of its fanout PLAs so that the original PLA can be eliminated. Since collapsing some PLAs into their fanout PLAs might preclude the possibility of collapsing other PLAs, they used the empirical results shown in [31] that suggest that smaller PLAs should be collapsed first (smaller in terms of inputs * product terms). The second packing operation attempts to merge PLAs that share a large number of inputs. So for each PLA, a list is formed containing the other PLAs that it shares inputs with (in decreasing order of inputs shared), and the algorithm attempts to merge them. Applying packing to the circuit from Figure 17 results in a reduction from 9 clusters to 5 clusters, and gives us our final network for 3-3-2 PLAs, shown in Figure 18.

PLAmap showed good mapping results when compared to preexisting technology mapping algorithms. TEMPLA was the best academic technology-mapper for CPLDs before PLAmap was created, but PLAmap was able to reduce the mapping depth (and therefore the delay) of TEMPLA by 50% at a mere 10% cost in area. PLAmap was also compared to Altera's MAX+PLUS II tool, which uses 12% less area than PLAmap but incurs 58% more delay. These results help justify our use of PLAmap as our technology-mapper.

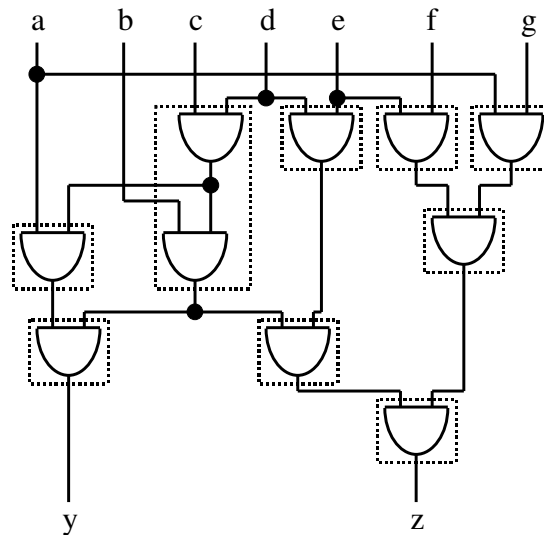


Figure 17. Sample circuit after PLAmap's mapping stage. Dashed boxes depict the clusters

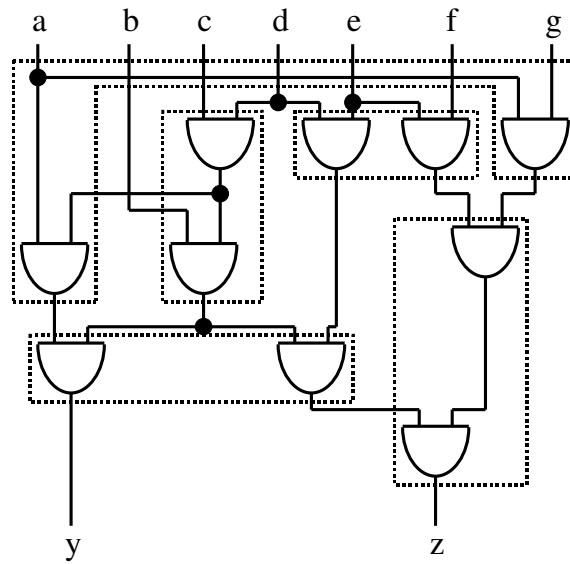


Figure 18. Sample circuit after PLAMap's packing stage. Dashed boxes depict the clusters, and ultimately the PLAs

4 Reconfigurable Hardware in SoC

This chapter provides information about the System-on-a-Chip design methodology, and presents ways in which reconfigurable hardware can be introduced into the SoC environment, citing examples of existing work. The chapter then concludes with some design examples that motivate the incorporation of reconfigurable hardware in SoC devices.

4.1 System-on-a-Chip Design Methodology

The System-on-a-Chip (SoC) design methodology has only recently become viable, due to the ever-increasing device densities realizable in VLSI systems. This methodology has many benefits, including reduced area, reduced delay, reduced power, and increased inter-device communication bandwidth. These advantages come at the cost of design complexity, however. Instead of dealing with just a few million transistors, a design team might now have to lay out hundreds of millions of transistors in order to form a complete SoC design. This difficult task would lead to very expensive devices, in terms of both time and money, if everything on the chip needed to be painstakingly laid out by hand.

One way that designers get around this complexity is by using hardware description languages (HDLs) and synthesis tools. In this process, a designer would write HDL code to describe his design, and would then use a synthesis tool to map that design to a gate-level description. This gate-level description would then be implemented in standard cells and laid out on the chip. This design methodology provides fast turn-around time, but the implementations suffer area, power, and delay penalties due to the intrinsic overheads of standard cells. In many cases these penalties are acceptable, but sometimes

they are not. An emerging process that can remedy some of these performance issues is the concept of intellectual property (IP) reuse.

The basic idea of IP reuse is that once a subcircuit is carefully designed, tested, and verified, that the next user who wishes to use the subcircuit won't have to go through those steps again. Thus an SoC designer could grab the layouts of components that have already been made (processors, DSPs, memories, etc.) and use custom logic or standard cell logic to integrate them together onto a single die. This would dramatically reduce the design time, not to mention that the verification of each component has already been done. The functionality of the integrated SoC system will still need to be verified, but this is true of any design.

4.2 Reconfigurable Hardware in SoC

Reconfigurable hardware can be integrated into an SoC device using either of the above methods: by describing the reconfigurable fabric in a HDL and using standard cells, or by integrating reconfigurable IP into the chip. HDL entry is typically easier to incorporate into the design flow since other aspects of the SoC will probably be described by HDLs as well, but using IP will provide better performance characteristics since the IP core has already been meticulously laid out. Clearly, a trade-off exists between ease-of-use and performance.

A third way to get reconfigurability onto an SoC is to simply use a large reconfigurable device as your SoC device. As we'll illustrate, there are chips currently being marketed with this exact goal in mind, and they are very capable of handling many SoC designs by themselves.

4.2.1 Using HDLs

Two examples of using HDLs to create reconfigurable hardware for SoC devices have come from the University of British Columbia. In [32] and [33], Wilton et. al. propose both LUT based and product-term-based reconfigurable architectures to be used in SoC applications.

In [32], they outline the details of the process that a chip designer would go through when using their logic. First, the SoC designer partitions the chip design into functions that will use fixed logic (which he describes in a hardware description language or HDL) vs. functions that will use programmable logic. He then acquires an HDL description of the behavior of the programmable logic core, provided by the UBC tools. The designer can then merge the HDL descriptions of the fixed and programmable logic and go through ASIC synthesis, place, and route tools in order to implement the HDL description. The chip is then fabricated, after which the programmable logic can be configured to behave however the designer chooses.

The proposed reconfigurable architectures in [32] and [33] are all directional, meaning that there are no feedback loops: this is a byproduct of the fact that they are creating their cores using HDLs. The need for directionality comes from the fact that many synthesis tools have problems with combinational loops. In [32], their directional requirement led them to create two types of implementations, a standard island-style architecture with directional routing and a gradual architecture, both of which used 3-LUTs (look-up tables) as their logic elements. An n-LUT is a user-configurable memory element that accepts n inputs and creates one output. These architectures are shown in Figure 19.

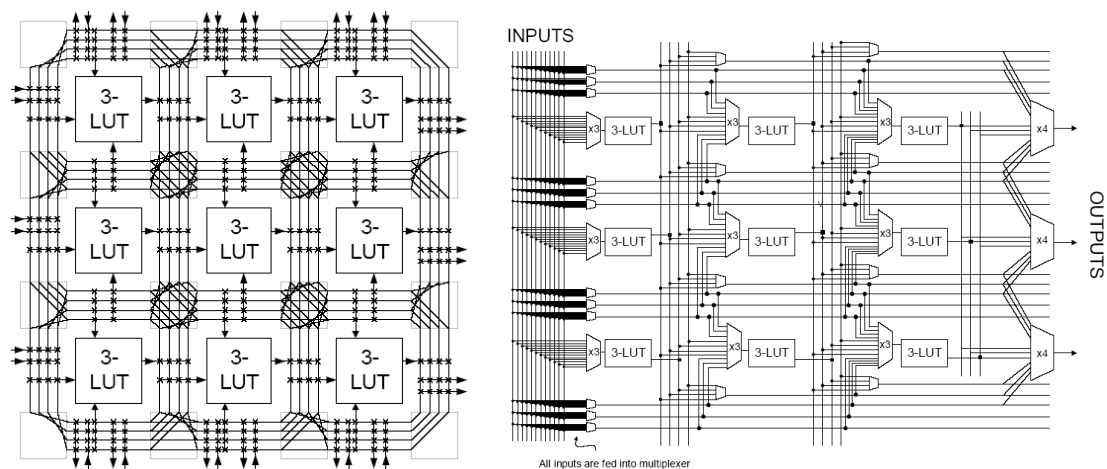


Figure 19. Directional (left) and gradual (right) architectures from [32]

Results from [32] showed that the gradual architecture performed better than the island-style architecture by 15% to 20%, and they also found that the gradual architecture works better if it is not rectangular, but rather it tapers to a slightly smaller (triangular) structure as you move along the datapath. These ideas helped guide the work in [33] in which they develop directional architectures that use PLAs as logic elements instead of 3-LUTs. For their PLAs, they used only the gradual architecture but again considered rectangular and triangular designs (shown in Figure 20). Their exploration of PLA-size showed that HDL cores with 9-input, 3-output, 9 or 18 product-term PLAs showed 35% area improvements and 72% speed improvements over their LUT-based architectures from [32]. They also showed that a triangular gradual architecture performs better than a rectangular architecture for PLAs.

UBC's deliverable is an HDL description, or "soft core", of the reconfigurable logic. Supplying a soft core would most likely make integration easier for the SoC designer, as the same synthesis tools can be used to create the reconfigurable logic as are used to synthesize the fixed portions of the chip. The synthesis of the reconfigurable logic would most likely be done using standard cells, however, which are very inefficient at creating reconfigurable architectures. Simple switches (which are often just single transistors), SRAM cells, and PLA arrays (if they choose a product-term style) don't exist in standard cell libraries, but they are very prevalent in reconfigurable architectures, and instantiating them in standard cells would create very drastic area, delay, and power penalties. The group admits to this problem, and suggests that the area, delay, and power penalties imposed by their soft cores would most likely only be acceptable for small amounts of programmable logic.

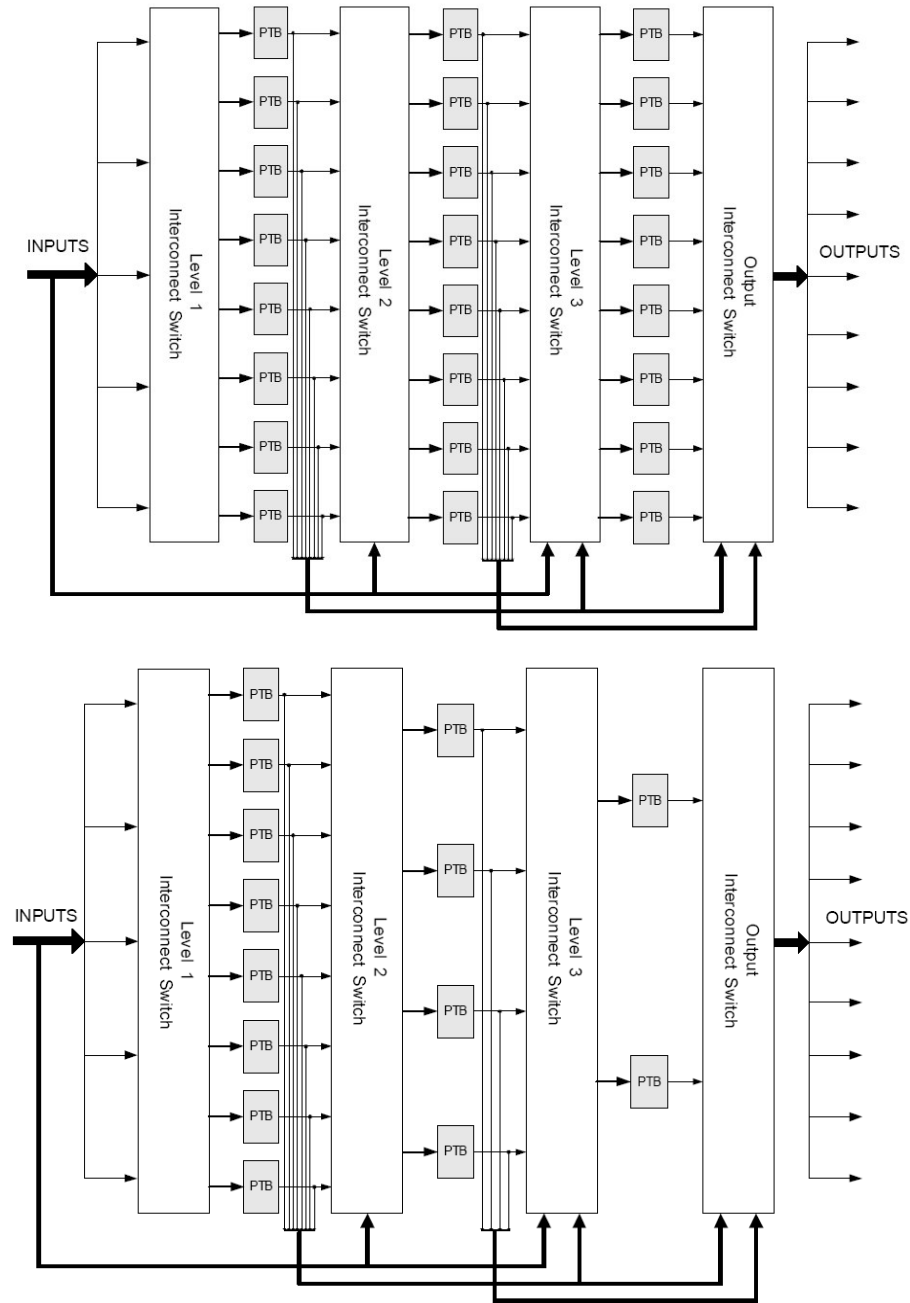


Figure 20. Product-term-based synthesizable architectures [33]

4.2.2 Using Reconfigurable IP

One example of reconfigurable IP comes from Mo and Brayton, who propose a highly regular “Glacier” PLA (GPLA) structure [34] for use in SoC devices. Their

proposal is to stack multiple configurable PLAs in a uni-directional structure using fixed river routing to connect them together. Optional registering can be implemented in the silicon below the river routing in order to support sequential designs. The result is a structure that benefits from both high circuit regularity and predictable area and delay formulation. This structure is shown in Figure 21.

GPLAs are very regular, and their authors site this as an advantage when transforming layouts to masks, as fewer layout patterns need to be examined. Fixed routing resources make the job of the CAD tools a bit more difficult by adding more restrictions, but this is partially alleviated by the fact that most input and output tracks in their PLAs are interchangeable – the only differences are that some input columns feed directly through to input columns on the next PLA. In their results, they display that GPLAs and Xilinx XC4000XL FPGAs with similar programmable bit counts can support roughly the same number of designs, showing that their densities in terms of gate-count per programmable bit are similar.

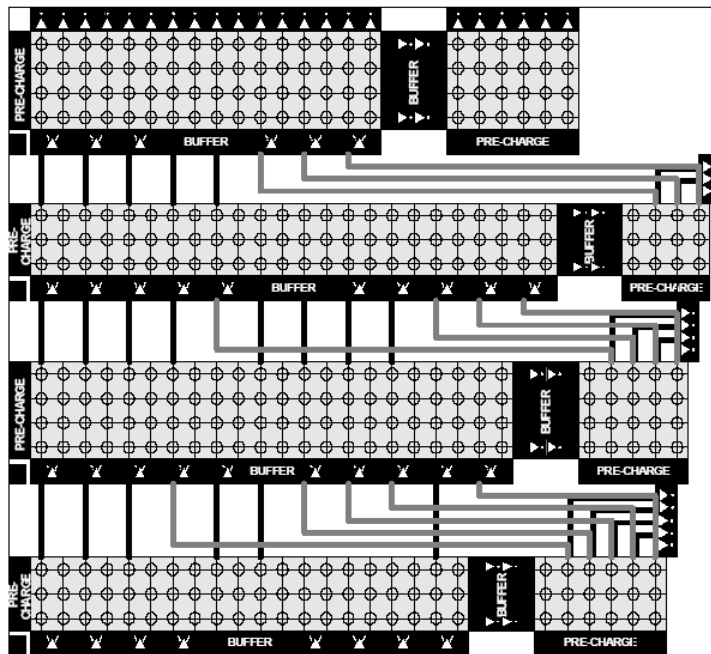


Figure 21. Glacier PLA (GPLA) with configurable PLAs and fixed river routing [34]

Another work presents a high-performance programmable logic core for SoC applications [35]. In this paper, a new architecture is developed which uses a high-performance dynamic circuit design style called OPL. OPL is a precharged-high logic family, so only discharging is necessary upon function evaluation. Due to this, they show that OPL designs provide 5x speedups over conventional circuit design styles when implementing circuits that map well to wide NOR gates. The work also introduces a novel product-term-based logic structure that utilizes OPL-friendly NOR gates. While most product-term-based reconfigurable logic provides a fixed input, product term, and output capacity, the logic structures proposed in this paper provide further gains by allowing these amounts to be variable. This avoids the area losses caused by unutilized logic resources in most product-term-based designs, because most PLAs are only partially utilized.

One important consideration for this OPL-based design is clock distribution, as the logic family requires successive clock phases to be present with very short separation times. This requires considerable clock-generation overhead, which takes up area that could otherwise be utilized for logic. Power consumption is also increased due to the need for a large number of minimally spaced clocks. The goal of this device, though, was increased speed. A test chip was produced and timing values extracted, and results showed that this new architecture provided an average speedup of 3.7x over a Xilinx Virtex-E FPGA.

The previous examples were academic in nature, but commercial reconfigurable IP cores exist as well. Actel, for example, has an embedded FPGA core that can be tiled into SoC devices to provide anywhere from 5000 to 40,000 equivalent ASIC gates [36]. They also have tile configurations that provide cascadable RAM modules for applications that require more memory. Elixent, another company creating reconfigurable IP cores, has developed what they call a D-Fabrix Processing Array [37]. A D-Fabrix tile contains two 4-bit ALUs, two registers, and two switchboxes. By tiling these units in the hundreds or thousands, the array is capable of efficiently supporting algorithms that

require high arithmetic throughput. Their reconfigurable IP is so useful that it is already being licensed by companies for use in SoC applications [38].

4.2.2.1 Totem

The D-Fabrix Processing Array is an example of a reconfigurable fabric that is tailored to a particular application domain: in this case, arithmetic designs. D-Fabrix already exists, so it can easily be integrated into new SoC design starts. If an SoC designer wanted a reconfigurable fabric that was tailored to some other domain, however, the time required to design the domain-specific reconfigurable fabric would probably be too time-prohibitive for the designer to accept. Quick turn-around time is an important constraint for almost all designs, and should be considered when trying to create new reconfigurable fabrics. The Totem project at the University of Washington is an attempt to create these domain-specific reconfigurable architectures quickly and automatically, such that there is no negative impact on the SoC's design cycle.

Previous work in Totem [6-25] leveraged the RaPiD [1, 2] architecture, which was developed by Carl Ebeling et. al. at the University of Washington. RaPiD is an architecture that targets the digital-signal-processing (DSP) domain. It is a one-dimensional word-wide architecture that uses coarse-grained units (ALUs, Multipliers, RAMs, etc.) to perform computations. A picture of this is shown in Figure 22.

Data flows horizontally in the array, switching onto vertical wires to reach functional units and then returning to horizontal wires. The routing structure is composed of three types of tracks: feedback tracks which route a unit's output back to its inputs, short tracks that span a small number of functional units, and long tracks that span a large number of functional units. Bus connectors exist on the long tracks in order to allow even longer tracks to be formed, and also to provide registering in the interconnect. The datapath is 16-bits wide, leveraging the fact that DSP computations such as multiplication and addition tend to operate on word-wide data rather than bit-wide data.

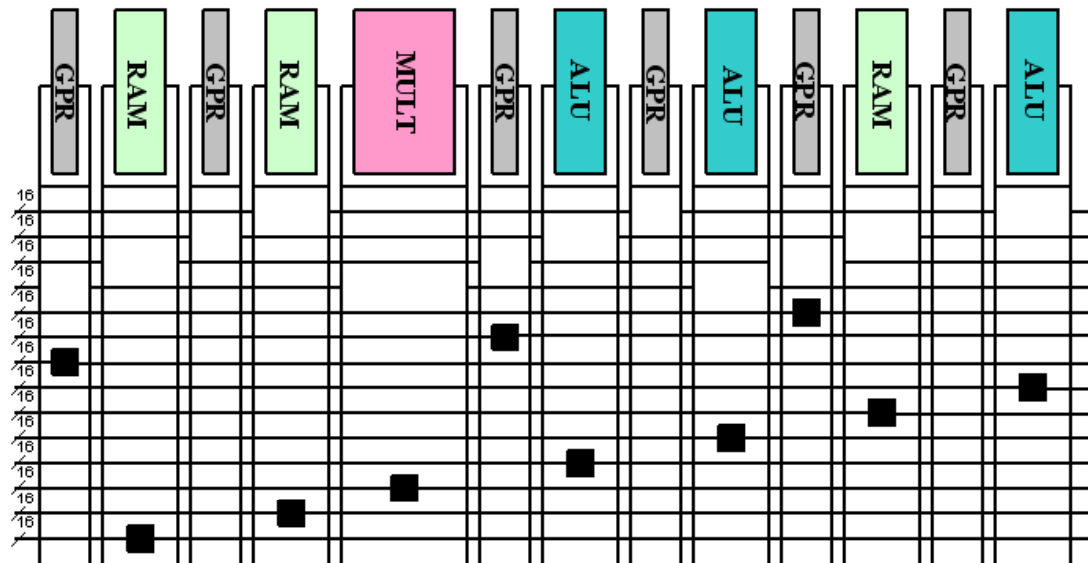


Figure 22. A tile in the RaPiD array [1, 2]

The RaPiD architecture is tailored to be domain specific by adjusting the blend of resources that it contains. Many modifications can be made, including varying the amount and mixture of functional units, the bit-width of the datapath, the number of data lines, the mix of long/short wire segments, and the number of registers in the datapath. New functional units can even be created in order to target RaPiD-style architectures to non-DSP applications, as was demonstrated by RaPiD-AES [12], which created a new set of private-key encryption based functional units for targeting the algorithms from the Advanced Encryption Standard competition [49]. Different algorithms will clearly require different resource mixes, and Totem is able to provide this by modifying the RaPiD architecture to meet the algorithm's needs.

The basic flow of Totem-RaPiD is shown in Figure 23. The SoC designer supplies the Architecture Generator with a domain description, including netlists and constraints. The Architecture Generator uses this information to create an architecture description in a hardware description language (HDL) that is sent to the VLSI Layout Generator and the Place & Route Tool Generator. The VLSI Layout Generator creates an actual layout mask for the described architecture and gives this layout to the designer. This layout is what will physically be put onto the chip. The Place & Route Tool Generator creates the

physical design tools for the specified reconfigurable architecture. The tools created here are responsible for mapping user designs to the architecture.

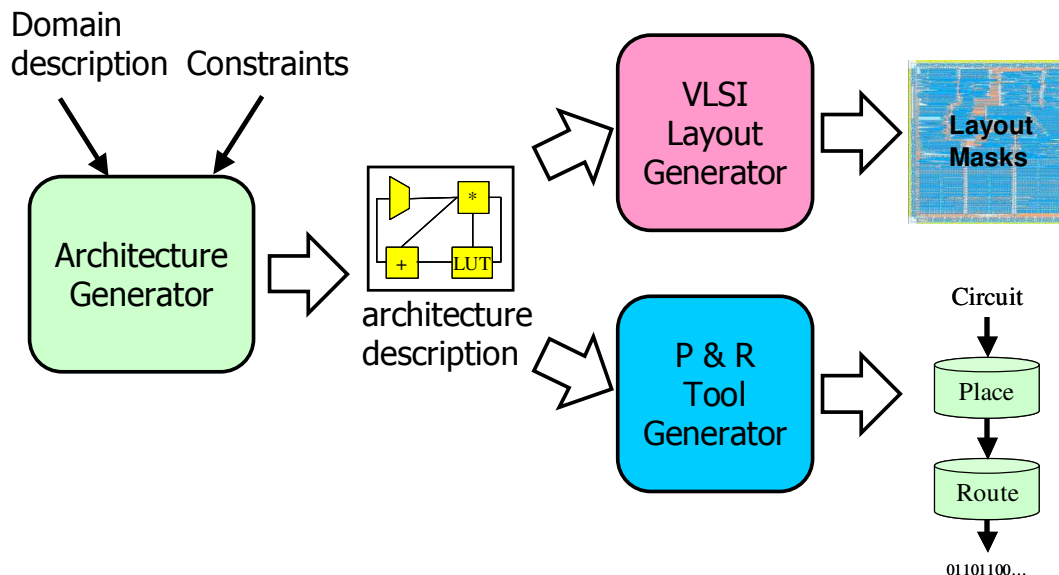


Figure 23. The Totem-RaPiD tool flow

Work in Totem-RaPiD demonstrates that domain-specific reconfigurable architectures can be developed quickly and automatically in response to the input of a domain specification. It is hoped that SoC designers will be more likely to utilize reconfigurable hardware if it can be developed quickly and tailored to the specific domain of the SoC. Totem provides these attributes.

4.2.3 Reconfigurable Chips as the SoC

A third option, and one that is becoming more and more viable, is the use of an FPGA as a configurable-system-on-a-chip (CSoC). Instead of putting reconfigurable hardware onto an SoC device, we are now implementing the SoC device totally in reconfigurable hardware.

The XILINX Virtex-II Pro series of FPGAs are a good example of this configurable-SoC idea. The Virtex-II Pro (XC2VPX70) [39] has over 74000 LUTs, two IBM PowerPC processor blocks, over 300 18 x 18 bit multiplier blocks, over 5 megabytes of

block RAM, and 20 multi-gigabit transceiver blocks. Such a large, dynamic device is capable of handling many SoC designs by itself.

Xilinx has further targeted the CSoC market with the recent introduction of their Virtex-IV devices. Each of these devices comes tailored to a specific application domain, including signal processing, embedded processing/high-speed connectivity, and high-performance logic. Among the three domains there are seventeen available devices, allowing the SoC designer to choose the product that best matches the size and resource requirements of their design. As an example of how these devices are tailored to a domain, the signal processing devices contain extra memory and DSP specific slices that “support over 40 dynamically controlled operating modes including; multiplier, multiplier-accumulator, multiplier-adder/subtractor, three input adder, barrel shifter, wide bus multiplexers, or wide counters.” [40]

4.3 Motivating Reconfigurable Hardware in SoC

Reconfigurable hardware has many uses in the field of Systems-on-a-Chip. The reconfigurable fabric can be used as a coprocessor to speed up computation, whether by exploiting parallelism in code or targeting frequently executed loops. Reconfigurable resources can also be used for supporting different standards/protocols, or for providing easy upgradeability. The reconfigurable resources can even be used to test other parts of the SoC device.

A coprocessor design that is targeted for SoC is described in [41]. The work targets motion estimation, which is a complex computation that is found in video compression algorithms such as MPEG-4. Their system is a chip that consists of a Digital Signal Processor (DSP) and a number of reconfigurable arrays, with a bus providing communication between them, and a controller integrating the elements together. During runtime, the controller identifies algorithms that can be efficiently executed in hardware and it dynamically programs the reconfigurable blocks to implement these algorithms. The use of a coprocessor provides speed and power gains over a lone DSP, and their

results show that the mating of DSP and reconfigurable logic provide 4x power gains and 2x area gains over an FPGA implementation using a Xilinx Virtex-E.

Another use of reconfigurable logic in SoC is to allow conformity to different standards. This could be especially useful in a domain like wireless communication, where several different communication protocols are in use. Cell phones can operate using either analog or digital service, and in the digital realm there are different transmission technologies that a phone can use (TDMA/GSM or CDMA for example). Because of these different technologies, a phone that works in the United States won't necessarily work in Europe, but a cell phone that contained a chip with reconfigurable logic could use this logic to adjust its transmission protocols according to what is required in the specific region. A phone could then conform to all wireless networks without requiring hardware that is specifically dedicated to each protocol.

Along a similar line, reprogrammability allows upgrades to be made very easily. If a chip uses a protocol that is likely to be upgraded or improved upon soon, it can be implemented in programmable logic. As long as there is enough programmable logic available to accommodate the new upgraded protocol, the chip will be able to accommodate the change and continue functioning in its environment. This will save the cost of having to fabricate a whole new device simply in order to support a change in protocol or procedure.

Reconfigurable logic can also be included in an SoC in order to facilitate testing. As described in [42], having an embedded FPGA on an SoC allows for built-in-self-test (BIST) to occur without any overhead. They cite that logic BIST for an ASIC can require a 20% area overhead and discernable performance degradation. But by implementing the BIST logic on the FPGA during testing one can eliminate these overheads, and once the testing is done the BIST logic can be completely removed and replaced by functions that will be useful during normal operation. The work describes both how to test the embedded FPGA and how to then use the embedded FPGA to test the other cores on the chip.

A second work also considers using reconfigurable logic to test an SoC device [43]. In this work a flexible network is integrated on the SoC, and is used to select internal signals that are of interest for specified debug tasks. These signals are then provided to a programmable logic core (PLC), which is used to implement debugging circuitry. Tests can be controlled using an on-chip processor if one is present; otherwise they are controlled using an external JTAG interface. While the testing methods proposed in this work use the reconfigurable logic solely for debug purposes, they predict that this debug hardware will require less than 10% of the area of the SoC device. Considering the improved observability and controllability that this process will provide, these area costs seem quite reasonable.

These are just a few of the possible uses of reconfigurable hardware in SoC devices. The work described in this paper is an effort to create reconfigurable hardware specifically for SoC by tailoring the fabric to the SoC's specific domain, and by automating the hardware generation process so that SoC designers don't have to worry about adverse effects on their design cycle.

5 Research Framework

This dissertation presents tools that automate the creation of domain-specific CPLDs for System-on-a-Chip. In order to develop and evaluate these tools, we first needed to acquire the circuits that would be used in our test domains. We also had to create the area and delay models that would be used for evaluating our architectures. This chapter presents these items in more detail.

5.1 Totem-PLA

In Chapter 6 we will begin our exploration of the domain-specific CPLD space by first considering domain-specific PLAs and PALs, a project termed Totem-PLA. In order to create these structures, we first had to acquire appropriate circuits and group them into domains. For each of these domains, we could then create domain-specific PLAs and PALs. Additionally, these PLA and PAL architectures had to be evaluated for performance, which required the creation of accurate delay models.

5.1.1 Circuits

Individual PLAs and PALs are logic devices that are capable of picking up registers only at their outputs, not internally. As such, sequential circuits cannot be mapped to individual devices unless some sort of feedback is implemented and the arrays are utilized for multiple passes of logic. This is not typically done, as it is much easier to connect multiple PLAs/PALs into a CPLD structure that provides the desired behavior with better performance. Because of this characteristic of PLAs and PALs, only combinational circuits are used in Totem-PLA.

The first source of circuits for Totem-PLA was the ESPRESSO suite [44]. ESPRESSO is the standard two-level logic minimization algorithm in use today, and the circuits in the suite are the same ones that the ESPRESSO algorithm used for testing. A

second set of circuits came from the benchmark suite compiled by the 1993 Logic Synthesis Workshop, also called LGSynth93 (the well known MCNC benchmark circuits also come from this suite) [45]. As a whole, these circuits are commonly used in research on programmable logic arrays. The circuits are already in PLA format, so they need no manipulation before being sent through the tool flow presented in Chapter 6.

Table 1 gives information on the circuits used in Totem-PLA. The function of many of these circuits is unknown, so we were unable to group them into domains according to their target applications. For the purposes of this work, it was sufficient to group them into domains according to relative size, as this will be a factor in how well our subsequent algorithms perform.

Table 1. The circuits used for Totem-PLA

Domain	Circuit	IN	OUT	PT	Prog. Conn.
1	ti	47	72	213	2573
	xparc	41	73	254	7466
2	b2	16	17	106	1941
	shift	19	16	100	493
	b10	15	11	100	1000
	table5.pla	17	15	158	2501
	misex3c.pla	14	14	197	1561
	table3.pla	14	14	175	2644
3	newcpla1	9	16	38	264
	tms	8	16	30	465
	m2	8	16	47	641
	exp	8	18	59	558
4	seq	41	35	336	6245
	apex1	45	45	206	2842
	apex3	54	50	280	3292

The table shows the input, output, product term, and programmable connection count of each circuit used, as well as the domain groupings of the circuits. In sum-of-products notation, each occurrence of a variable is called a literal. For a PAL, the number of literals is equal to the number of programmable connections that are used in the array. For a PLA, since the OR-plane is programmable, the number of programmable connections in the array equals the number of literals plus the number of product terms. The table lists the number of programmable connections used in a PLA implementation. The number of programmable connections is similar to circuit size in that it will have an

effect on how well our algorithms perform, and it is therefore a consideration when grouping circuits into domains.

5.1.2 Delay Model

We developed a delay model in order to evaluate the performance of the PLA and PAL arrays: the model represents the propagation delay of a signal through the entire array. We define the propagation delay as the time between the input reaching 50% of VDD and the corresponding output reaching 50% of VDD: this is a common way to measure propagation delay.

In CMOS design, a simple RC model is often used to obtain first-order estimates of propagation delays. Using this model, a cutoff transistor is represented by an open circuit, and an active transistor is represented by a closed switch in series with a resistor. The delay is then based on the charging or discharging of some output capacitance in response to a change in input voltage. Figure 24 shows this for an inverter circuit.

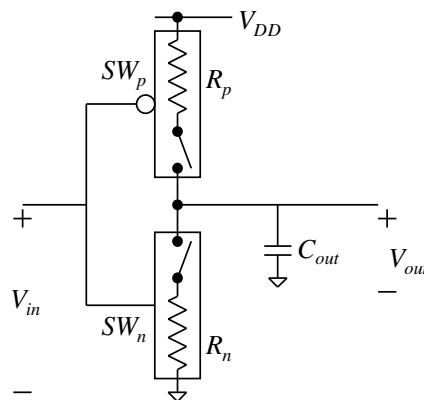


Figure 24. The RC model for an inverter [46]

Using this RC model, the propagation delay through a transistor can be estimated by Equation 2 (derivation found in [46]). In this equation, X is a constant, R is the resistance of the transistor, and C_{out} is the output capacitance. Since we will not be changing the sizes of any of the transistors in our arrays, the value of the resistance R is constant for a given transistor and can be absorbed into the constant X , giving us Equation 3. As this

equation states, the delay through any transistor in our PLA/PAL array is estimated to be linearly proportional to the output capacitance.

$$t_{prop} = X * R * C_{out} \quad (2)$$

$$t_{prop} = X * C_{out} \quad (3)$$

Figure 25 highlights the worst-case propagation path in one of our pseudo-nMOS PLAs. Notice that this figure differs from the previous pseudo-nMOS diagram, as there are now buffers inserted to help drive the AND-plane and OR-plane. All AND-plane array locations are laid out the same, and will contribute the same capacitance to the corresponding signal path. This is also true of the locations in the OR-plane. We are neglecting some very small capacitive variations that will be caused by differences in location within the AND-plane or OR-plane, but these are small enough to be insignificant. In Figure 25, the devices responsible for driving the signal path are numbered so that we can examine them each individually. By summing the propagation delays through each of these numbered items, we will obtain the propagation delay through the PLA. There will be two propagation delays of interest, one for a rising output and one for a falling output, and the analysis will be similar for each.

The PLA shown in Figure 25 has IN inputs, PT product terms, and OUT outputs. The propagation delay through the input buffer, labeled 1 in the figure, is shown in Equation 4. This buffer must drive the capacitance of one inverter (C_{inv1}), and PT array locations (each C_{loc}). The inverter, labeled 2 in the diagram, must similarly drive PT array locations (Equation 5). Next, either a pull-up or pull-down transistor (3a or 3b respectively) must charge/discharge IN array locations of C_{loc} , plus C_{buf} (Equation 6). The buffer, labeled 4, must drive OUT array locations of C_{loc} (Equation 7). For the OR-plane, again either a pull-up or pull-down transistor (5a or 5b respectively) must charge/discharge PT array locations, plus the output inverter C_{inv2} (Equation 8). The output inverter, labeled 6, must then drive some C_{out} that we have fixed to a constant value (Equation 9).

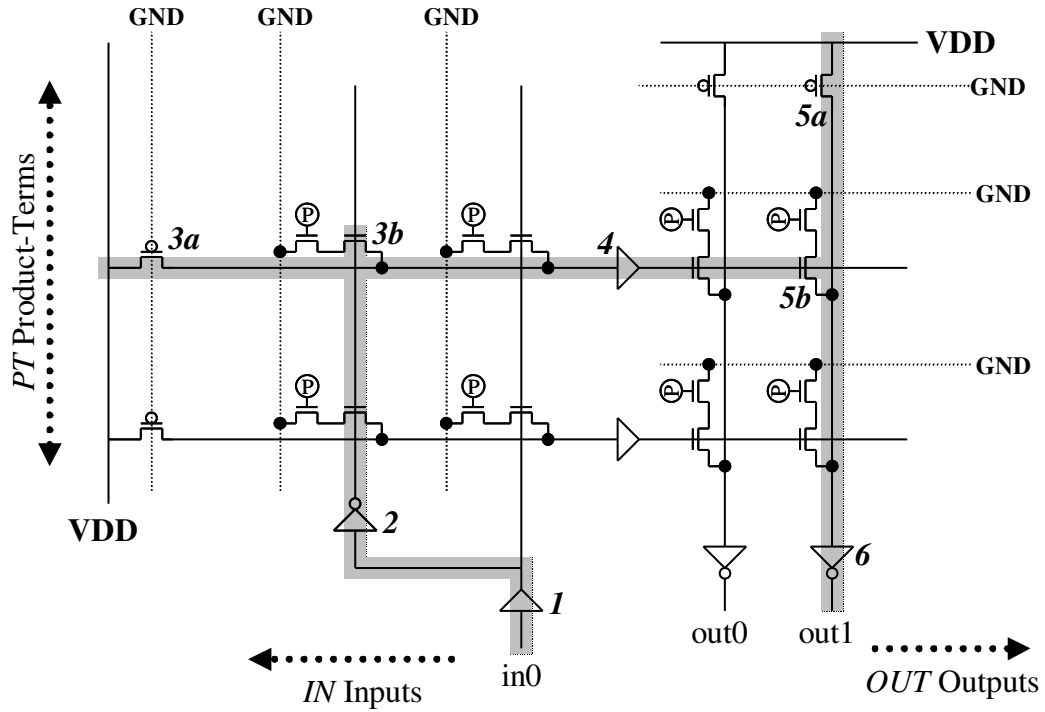


Figure 25. The worst-case propagation path through a PLA is highlighted, and the devices driving the path are number

$$t_{prop1} = X_1 * C_{inv1} + X_2 * PT * C_{loc} \quad (4)$$

$$t_{prop2} = X_3 * PT * C_{loc} \quad (5)$$

$$t_{prop3} = X_4 * C_{buf} + X_5 * IN * C_{loc} \quad (6)$$

$$t_{prop4} = X_6 * OUT * C_{loc} \quad (7)$$

$$t_{prop5} = X_7 * C_{inv2} + X_8 * PT * C_{loc} \quad (8)$$

$$t_{prop6} = X_9 * C_{out} \quad (9)$$

Equations 4-9 represent the propagation delays through elements in the PLA array, and can be summed to get the propagation delay through the entire PLA. We can do some simplification by recognizing that C_{inv1} , C_{buf} , C_{inv2} , and C_{out} are all independent of PLA size, and are therefore constants. Summing the equations thus gives us Equation 10. C_{loc} will also be a constant value, and this allows us to rewrite Equation 10 as Equation 11. Equation 11 displays that the delay through a PLA is linearly related to the number

of IN , PT , and OUT values in the device. We must now simply determine the constants in Equation 11 for the respective cases of a rising and falling output value.

$$t_{prop} = X_{10} + ((X_2 + X_3 + X_8) * PT + X_5 * IN + X_6 * OUT) * C_{loc} \quad (10)$$

$$t_{prop} = X_{11} + X_{12} * PT + X_{13} * IN + X_{14} * OUT \quad (11)$$

In order to acquire the constants in Equation 11, we used Cadence's layoutPlus to create 39 PLAs with IN , PT , and OUT values that varied from 1 to 80. These PLAs were then netlisted and simulated using hSpice, and the worst-case rise and fall times of the arrays were acquired. Using this data, we then performed a linear fit and acquired the constants in Equation 11. The models for the final rise- and fall-time propagation delays for the PLAs are shown in Equations 12 and 13.

$$t_{rise_PLA} = (305 + 15.3 * IN + 15.0 * PT + 7.0 * OUT) * 10^{-12} \text{ sec} \quad (12)$$

$$t_{fall_PLA} = (345 + 16.0 * IN + 15.2 * PT + 4.1 * OUT) * 10^{-12} \text{ sec} \quad (13)$$

We next compared the predicted results to the actual results we obtained in order to obtain the "error" imposed by these models. For the rise-time model, the worst-case error was 12.8%, the average magnitude of the error was 2.1%, and the standard deviation of the error was 3.0%. For the fall-time model, the worst-case error was 10.2%, the average magnitude of the error was 2.6%, and the standard deviation of the error was 3.5%. As can be seen, these models very accurately predict the timing through our PLA arrays.

Note that these equations are only valid because we are using constant device sizes in our PLAs. This allows us to keep our design complexity at a reasonable level, at a cost to the performance of our architectures. A production quality flow would use logical effort to size the devices in order to maximize performance, which would have a complicating effect on the area and delay models, and would also require a much more significant design effort in terms of the creation of VLSI layouts.

The simplification of fixed device sizing is valid for our work because we are interested in comparing the performance of domain-specific architectures versus domain-generic architectures rather than obtaining the highest performance possible. Since our fixed device sizing will affect the domain-specific and domain-generic architectures

similarly, our performance comparisons will still be accurate, despite the fact that the overall performance may be an order of magnitude away from optimal for larger architectures. The PLAs, PALs, and CPLDs that we create all use the same fixed device sizings.

Determination of the propagation delays for PALs was very similar to that for PLAs. The only difference is that PALs use fixed gates for the OR-plane (they actually use NOR-gates since we do NOR-NOR style), so we needed to remove the variable corresponding to output count and replace it with a variable that would take into account the largest NOR-gate used in the PAL.

We allow our PALs to use NOR-gates of any size since we implement them in pseudo-nMOS style, and this prevents us from having long charge/discharge paths. In the worst case, a NOR-gate has a single transistor charging or discharging the output node. Using the RC model, the output capacitance seen by the NOR-gate is roughly linear in the size of the gate, because the number of transistors attached to the output node is one more than the size of the NOR-gate (and any fanout and line capacitance will be constant for our case).

We determined the constants for our PAL devices in the same method as for our PLA devices. We created and simulated a total of 96 PALs, varying the IN and PT values from 1 to 80, and varying the NOR-gate size from 3 to 7. We then performed a linear fit to this data to acquire our constants. Equations 14 and 15 model the propagation delay for our PAL devices.

$$t_{rise_PAL} = (71 + 17.1 * IN + 6.3 * PT + 21.0 * MAX_NOR_SIZE) * 10^{-12} \text{ sec} \quad (14)$$

$$t_{fall_PAL} = (196 + 20.8 * IN + 8.1 * PT + 4.9 * MAX_NOR_SIZE) * 10^{-12} \text{ sec} \quad (15)$$

For the rise-time model, the worst-case error was 16.7%, the average magnitude of the error was 5.1%, and the standard deviation of the error was 6.4%. For the fall-time model, the worst-case error was 10.6%, the average error was 2.7%, and the standard deviation of the error was 3.6%. One interesting thing to note about these equations is

that the rise time is heavily related to the size of the largest NOR-gate, since a wide pseudo-nMOS NOR-gate will have a difficult time pulling up.

5.2 Totem-CPLD

In Chapter 7 we progress from domain-specific PLAs and PALs to domain-specific CPLDs. Just as with PLAs and PALs, we first had to acquire circuits and group them into domains. Using these domains, we created CPLD architectures that were evaluated for performance, which required the creation of both area and delay models.

5.2.1 Circuits

We use PLAMap as the technology-mapper in our flow for creating domain-specific CPLDs. While PLAMap's algorithms will work on any circuit whose largest gate fits within the specified PLA size, the program has been written such that it requires the input circuits to be 2-bounded. We are thus restricted to the use of 2-bounded BLIF format circuits (2-bounded means that no gates have more than 2 inputs).

Very few interesting circuits are available in BLIF format, so this necessitated some pre-processing steps. Circuits are most easily obtained in HDL (Verilog or VHDL) formats, so we developed a process that could take HDL files and transform them into 2-bounded BLIF files. This process is shown in Figure 26.

The HDL files are first loaded into Altera's Quartus 2 program. Quartus 2 performs synthesis on the files, and dumps the designs into BLIF format (developers at Altera were very helpful in providing this hidden functionality to us). SIS is then used to transform the BLIF file into a 2-bounded network using the *tech_decomp -a 1000 -o 1000* and *dmig -k 2* commands, followed by the *sweep* command.

Before the complete flow in Figure 26 was put together, we were performing preliminary tests using some small circuits from the LGSynth93 suite. The functions of these circuits are unknown, so we simply grouped them according to size into small, medium, and large domains. These circuits, along with their input, output, and 2-bounded gate count, are shown in Table 2.

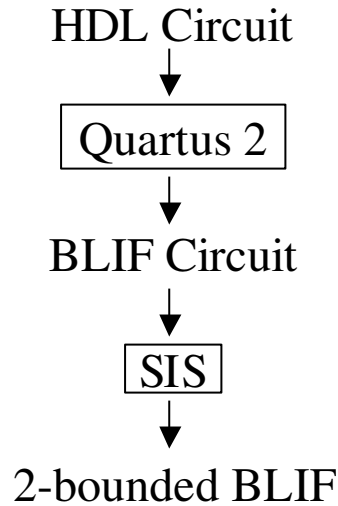


Figure 26. Flow for transforming circuits to BLIF format

Once the flow shown in Figure 26 was finalized, we used it to create five domains of circuits. Two of them, the combinational and sequential domains, consist of files gathered from the LGSynth93 Benchmark Suite. These circuits were acquired in BLIF format, and simply needed to be 2-bounded by SIS.

Table 2. The sample domains used for preliminary testing

small	IN	OUT	GATES	medium	IN	OUT	GATES	large	IN	OUT	GATES
misex1.blif	8	7	60	i4.blif	192	6	246	alu4.blif	14	8	2907
ldd.blif	9	19	106	rd73.blif	7	3	457	C7552.blif	207	107	2246
i1.blif	25	13	43	apex7.blif	49	37	253	x1.blif	51	35	1888
decod.blif	5	16	50	alu2.blif	10	6	455	cordic.blif	23	2	2814
pm1.blif	16	3	73	9sym.blif	9	1	395	i10.blif	257	224	2286
rd53.blif	5	3	101	C880.blif	60	26	352	pair.blif	173	137	1525

The remaining three domains consist of floating-point, arithmetic, and encryption files respectively. These files were accumulated from a variety of sources, including OpenCores.org [47], from Altera software developers, as Quartus 2 megafunctions, and from open source floating-point libraries [48]. All of these files were provided in HDL format, and went through the entire flow shown in Figure 26 in order to be used in our work.

The floating-point domain consists of several different units, including floating-point multipliers, adders, and dividers. Also included is an LNS divider, an LNS multiplier,

LNS and floating-point square root calculators, and a floating-point to fixed-point format converter.

The arithmetic domain consists of several different implementations of multipliers and dividers, as well as a square root calculator and an adder/subtractor. The encryption domain consists of the Cast, Crypton05, Magenta, Mars, Rijndael, and Twofish encryption algorithms (all sans memories), all of which were recent competitors to become the advanced encryption standard [49]. The domains are all summarized in Table 3.

Table 3. The main domains used in our work

Comb	IN	OUT	GATES	REGs
C1355	41	32	542	0
C17	5	2	8	0
C1908	33	25	460	0
C3540	50	22	1045	0
C432	36	7	175	0
C499	41	32	406	0
C5315	178	123	1978	0
C6288	32	32	2350	0
C7552	207	107	2246	0
C880	60	26	352	0
c8	28	18	219	0
cm138a	6	8	26	0
cm150a	21	1	46	0
cm151a	12	2	23	0
cm152a	11	1	31	0
cm162a	14	5	42	0
cm163a	16	5	42	0
cm42a	4	10	22	0
cm82a	5	3	22	0
cm85a	11	3	44	0
cmb	16	4	47	0

FP	IN	OUT	GATES	REGs
FPMult	65	35	9895	698
fpadd	44	22	2213	0
fpmul	44	22	3687	0
fpdiv	44	22	5505	0
fpsqrt	22	22	2675	0
lnsdiv	44	22	340	0
lnsmul	44	22	331	0
lnssqrt	21	22	24	0
float2fix	36	34	704	142
fp_mul	67	57	8500	174
fp_sub	67	35	1786	199
fp_add	67	35	1786	199

Seq	IN	OUT	GATES	REGs
s1196	15	14	481	18
s1238	15	14	552	18
s208.1	11	1	77	8
s344	10	11	122	15
s349	10	11	125	15
s382	4	6	150	21
s400	4	6	160	21
s420.1	19	1	165	16
s444	4	6	173	21
s526	4	6	241	21
s526n	4	6	272	21
s838.1	35	1	341	32
s953	17	23	369	29

Arith	IN	OUT	GATES	REGs
MultAddShift	32	32	4392	0
MultBooth2	34	33	759	37
MultBooth3	34	33	2238	36
MultSeq	34	33	529	53
serial_divide_uu	28	17	645	53
Adder	32	17	379	0
AddSub	33	16	370	0
Mult	32	32	4361	0
Div	32	32	3283	0
AbsValue	32	32	302	0

Enc	IN	OUT	GATES	REGs
cast	298	166	12934	301
crypton05	388	260	6980	261
magenta	452	387	4876	713
mars	389	172	23637	1071
rijndael	388	132	11618	261
twofish	261	196	14784	517

In chapter 9, we will consider the question of how to intelligently add resources to our CPLD architectures in order to support future circuits. The five main domains did

not provide as many data points as we desired, so we created an additional reduced domain from each of the originals. These reduced domains are shown in Table 4.

Table 4. Additional domains used for Chapter 9 results

CombA	IN	OUT	GATES	REGs
cm138a	6	8	26	0
cm150a	21	1	46	0
cm151a	12	2	23	0
cm152a	11	1	31	0
cm162a	14	5	42	0
cm163a	16	5	42	0
cm42a	4	10	22	0
cm82a	5	3	22	0
cm85a	11	3	44	0
cmb	16	4	47	0

SeqA	IN	OUT	GATES	REGs
s344	10	11	122	15
s349	10	11	125	15
s382	4	6	150	21
s400	4	6	160	21
s420.1	19	1	165	16
s444	4	6	173	21

FPA	IN	OUT	GATES	REGs
fpadd	44	22	2213	0
fpmul	44	22	3687	0
fpdiv	44	22	5505	0
insdiv	44	22	340	0
insmul	44	22	331	0

ArithA	IN	OUT	GATES	REGs
MultiBooth2	34	33	759	37
MultiSeq	34	33	529	53
serial_divide_uu	28	17	645	53
Adder	32	17	379	0
AddSub	33	16	370	0
AbsValue	32	32	302	0

EncA	IN	OUT	GATES	REGs
cast	298	166	12934	301
rijndael	388	132	11618	261
twofish	261	196	14784	517

5.2.2 Delay Model

Figure 27 represents the style of CPLD that our Chapter 7 tool flow creates, and the figure highlights the critical path in terms of propagation delay. The same analysis that was used to develop the delay model for the PLAs and PALs can be used to develop the delay model for our CPLDs. The main difference is that we now need to consider the interconnect network, as our path through the PLA must start and end in the CPLD interconnect, as shown in the figure.

The signal starts by being switched from the interconnect fabric into a PLA input track. This PLA input track is connected to each horizontal track in the interconnect. In the interconnect, there is one horizontal track for each primary input from I/O, and one track for each PLA output (of which there are $PLA_COUNT * PLA_OUT$). The signal then propagates through the PLA, as our earlier analysis showed. After this, the PLA must drive its output signal onto a vertical output track, which sees the same capacitive elements as the PLA input track. Finally, the signal must be driven onto the proper

horizontal track in the interconnect. This horizontal track is connected to every PLA input, of which there are $PLA_IN * PLA_COUNT$.

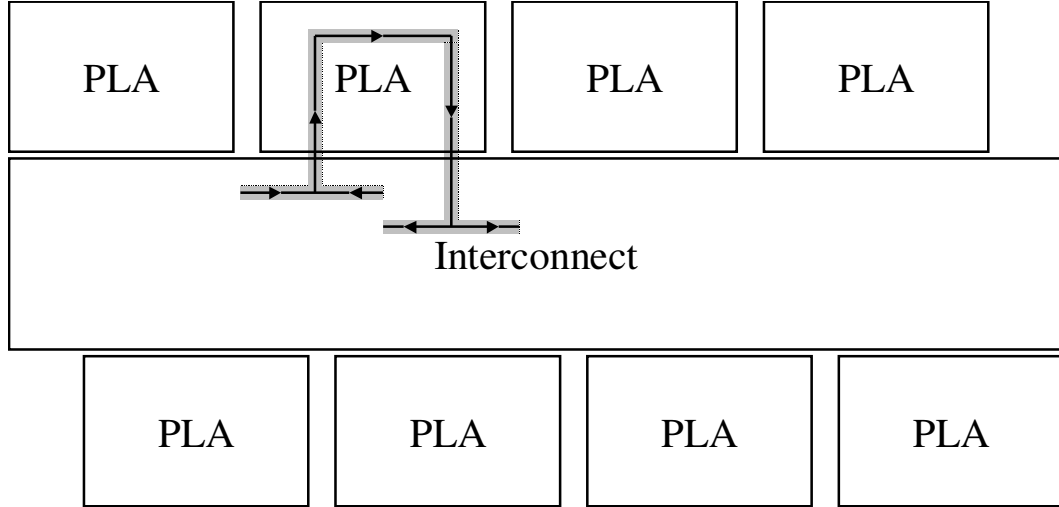


Figure 27. The delay through a PLA in our CPLD structure must start and end in the interconnect

The delay model for our CPLDs needs to account for these new terms. We developed the models in the same fashion as for the PLA and PAL arrays. We created 23 realistic CPLD architectures, varying IN from 2 to 12, PT from 2 to 36, OUT from 2 to 9, PLA_COUNT (CT) from 2 to 9, $PRIMARY_INPUT_COUNT$ (PI) from 2 to 18, then performed linear fits to the results. The linear fit equations are shown in Equations 16 and 17, describing the propagation delay of a signal that starts in the interconnect, traverses through a PLA, and returns to the interconnect.

$$t_{rise_CPLD} = (928 + 47.6 * IN + 3.7 * PT + 13.4 * OUT + 18.2 * PI + (8.63 * OUT + 32.5 * IN) * CT) * 10^{-12} \text{ sec} \quad (16)$$

$$t_{fall_CPLD} = (867 + 46.0 * IN + 26.5 * PT + 8.7 * OUT + 5.4 * PI + (.5 * OUT + 23.8 * IN) * CT) * 10^{-12} \text{ sec} \quad (17)$$

For our rise-time model, the worst-case error was 7.4%, the average magnitude of the error was 2.0%, and the standard deviation of the error was 2.7%. For the fall-time model, the worst-case error was 8.5%, the average magnitude of the error was 2.1%, and the standard deviation of the error was 3.1%.

5.2.3 Area Model

Since we create the actual VLSI layout of the CPLDs that we specify, we can determine the exact area required for the device. The area that we report for the CPLDs is the area of the smallest rectangle that completely encompasses the CPLD architecture along with any additional logic required for programming the memory elements in the architecture.

6 Domain-Specific PLAs and PALs

The goal of our work is to automate the creation of domain-specific CPLD architectures, where we define a CPLD as a collection of PLAs or PALs that are interconnected by a crossbar. Using this simple definition, we can pinpoint two obvious ways in which we can tailor a CPLD to a domain: by creating domain-specific PLAs/PALs, and by creating domain-specific crossbar structures. This chapter addresses the first of these two options, detailing our work involving the automatic creation of domain-specific PLAs and PALs.

We are ultimately examining these PLAs and PALs in order to determine how they can be best utilized in a domain-specific CPLD that is designed for SoC applications. To achieve this, we decided to examine the usefulness of single PLA and PAL devices within the same framework – by trying to tailor the arrays to a specific domain for use in SoC. By examining the ways in which we can tailor individual PLAs/PALs to a domain, we are providing a building block for our domain-specific CPLD explorations as well as determining the feasibility of using PLAs and PALs as stand-alone reconfigurable structures in SoC designs.

6.1 Tool Flow

The domain-specific PLAs/PALs are created using the flow shown in Figure 28. The input from the customer is a specification of the target domain, containing a set of circuits (in .pla format) that the target architecture must support. In addition to the circuits, there may be a combination of delay or area requirements that the architecture will need to meet.

The circuits are first processed by ESPRESSO in order to minimize the number of product terms and literals that they contain. This allows us to implement the circuits

using less silicon. The resulting minimized circuits are then fed into the Architecture Generator, which attempts to create the smallest single PLA or PAL array that is capable of supporting every circuit. Only one circuit must be supported at any given time. The Architecture Generator outputs information specifying the chosen PLA or PAL array, and also provides configuration files for configuring each circuit on the specified PLA or PAL. Additionally, any delay or area requirements provided by the customer can be checked after the Architecture Generator creates the array, as we have accurate models for calculating array delay and area.

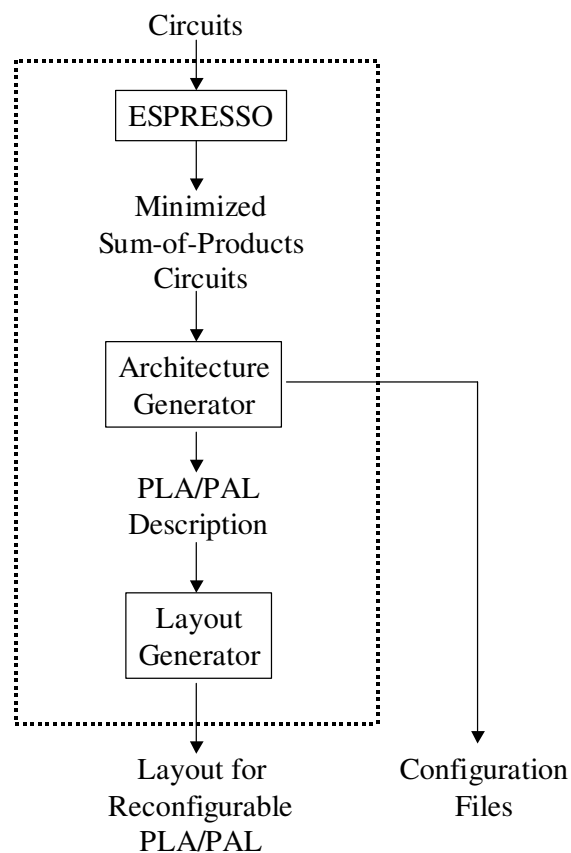


Figure 28. PLA/PAL Generation Tool Flow

After the Architecture Generator creates an array specification, the specification is fed to the Layout Generator, which creates a full layout of the array in the native TSMC .18 μ process. This layout includes the PAL/PLA array as well as the hardware necessary for programming the array.

6.2 Architecture Generator

The Architecture Generator must read in multiple circuits and create a PLA/PAL array capable of supporting all of the circuits. The tool is written in C++.

The goal of the Architecture Generator is to map all the circuits into an array that is of minimum size and which has as few programmable connections as are necessary. For a PLA, minimizing the number of inputs, outputs, and product terms in the array is actually trivial, as each of them is simply the maximum occurrence seen across the set of circuits. For a PAL we minimize the number of inputs and outputs the same way as for a PLA, and we minimize the number of product terms in the array by making each output OR gate as small as is possible.

Having minimized the number of inputs, outputs, and product terms in the array, the next goal is to minimize the number of programmable connections that are necessary in order to support each circuit. This is where we need an intelligent algorithm.

Figure 29 displays the problem that we face when trying to minimize the number of programmable connections that are necessary in the array. In this example we are trying to map two circuits to the same array (for the sake of this example the circuits, grey and black, implement the same function). A random mapping of the product terms (Figure 29, left) is shown to require 23 programmable connections, while an intelligent mapping (Figure 29, right) is shown to require only 12 programmable connections - a 48% reduction.

In this simple example the circuits happen to be the same, so we were able to obtain a perfect mapping. Circuits that are not identical will also have optimal mappings, which will result in a reduced number of programmable connections. Having fewer connections will allow us to compact the array in order to save area, and it will lower the capacitance, which will make our array faster.

The optimal mapping of product terms for two circuits can be found fairly efficiently. We first apply a cost of 1 to locations where we require a programmable connection, and a cost of 0 to locations where we do not require programmable connections. This accurately represents our problem because more programmable

connections will yield a higher cost – which directly represents the higher area and delay values that the array will produce.

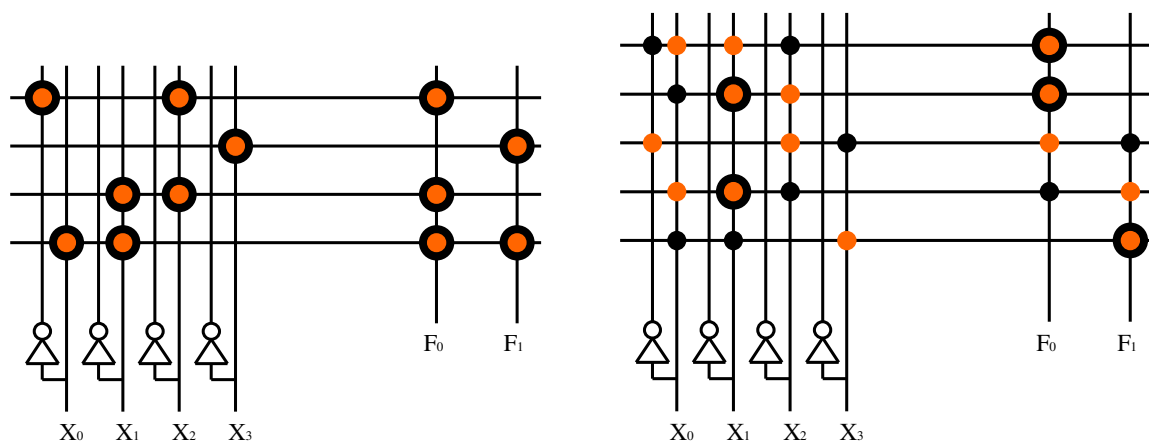


Figure 29. The left PLA shows two circuits mapped randomly, requiring 23 programmable connections. On the right they are mapped intelligently, requiring only 12 connections

Using this 0/1 cost model, the cost of each possible product-term matching between circuit A and B equals the number of programmable connections required by the pairing. We must now simply pair each of the m product terms from circuit A with a product term in circuit B such that we minimize the overall cost. This will give a mapping that uses as few programmable connections as possible: an optimal mapping.

This problem is equivalent to the “Optimal Assignment Problem”, and an algorithm developed by Kuhn and Munkres can be used to find the optimal assignment in $O(m^4)$, where m is the number of product terms [50] (see Appendix A for treatment). This algorithm works for two circuits, but our problem is to map n circuits onto the array, and n is likely to be greater than two. Our literature searches have found no algorithms that can efficiently find the optimal matching given more than two circuits.

One possibility, however, is to perform the Kuhn/Munkres algorithm on two circuits at a time, and to use a tree structure to combine the mappings. This can be done a couple ways, as shown in Figure 30. In this example we show two different ways of mapping four circuits ($N0 - N3$) together. The greedy nature of this formulation, however, is likely to provide poor mappings. An example of this is shown in Figure 31, which

displays a poor result that is obtained by using the Kuhn/Munkres algorithm in a tree fashion for three circuits that each have two product terms.

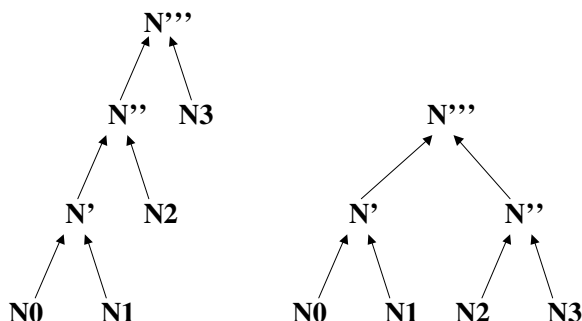
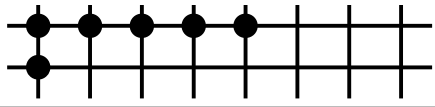
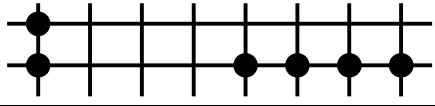
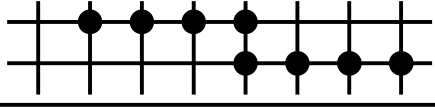
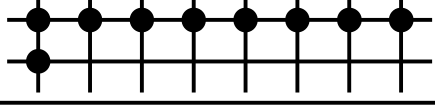
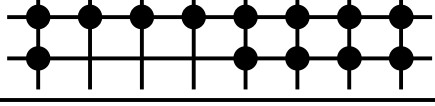
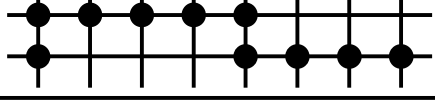


Figure 30. Tree options for using the Kuhn/Munkres algorithm

We have chosen not to implement the Kuhn/Munkres algorithm for mapping circuits to an array, largely because simulated annealing has proven to be very successful this application. The algorithm’s goal is to minimize the number of programmable connections required in the array. We define a basic “move” as being the swapping of two product-term rows within a circuit (we will introduce more complicated moves later), and the “cost” of a mapping is the number of programmable bits that it requires. The traditional annealing concept of a bounding box has no notion here, as our metric is not distance dependent, so any product term can swap with any other product term from the same circuit in a given move. For our annealing we use the temperature schedules published in [51].

The development of a cost function requires serious consideration, as it will be the only way in which the annealer can measure circuit placements. The previously mentioned cost function, in which we applied a cost of 1 to locations requiring a programmable bit and a cost of 0 to locations not requiring a bit, initially seems reasonable for our annealer. But looking deeper, the use of a simple 0/1 cost function would actually hide a lot of useful information from the annealer. The degree to which a programmable bit is required (how many circuits are using the array location) is also useful information, as it can tell the annealer how close we are to removing a programmable connection.

Circuit	Product Terms	Cost
N0		6
N1		6
N2		8
N'		9
N''		13
Best		10

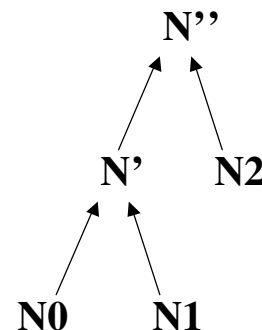


Figure 31. Suboptimality introduced by greedily combining the product terms in N0, N1, and N2 as shown. The greedy use of Kuhn/Munkres requires 13 programmable locations (N''), while the optimal solution (Best) uses only 10

Figure 32 displays this notion. In this example, we have one programmable connection used by two circuits and another connection used by five circuits. Both locations require a programmable bit, but it would be much wiser to move to situation A than to situation B, because situation A brings us closer to freeing up a connection.

The cost function that we developed captures this subtlety by adding diminishing costs to each circuit that uses a programmable connection. If only one circuit is using a connection the cost is 1; if two circuits use a connection it costs 1.5; three circuits is 1.75, then 1.875, 1.9375, and so on. Referring again to Figure 32 and using this cost function, moving to A is now a cost of -.45 (a good move) while moving to B is a cost of .19, which is a bad move. The cost function is shown in Equation 18, where x is the number of circuits that are using a position. As seen, each additional circuit that loads a position

incurs a decreasing cost, such that going from 7 to 8 is much cheaper than going from 1 to 2, for example.

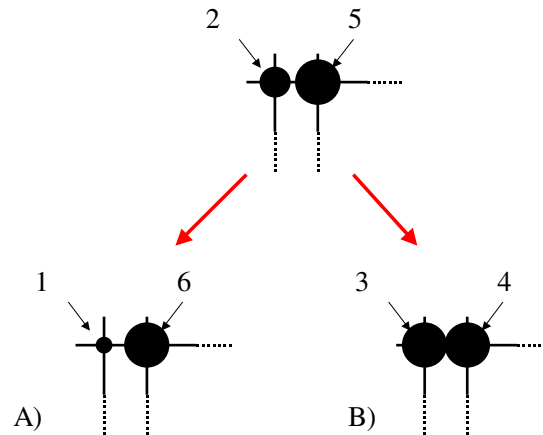


Figure 32. Moving to situation A puts us closer to removing a programmable connection

Because PALs and PLAs are structurally different, we need an annealing algorithm that can work on both types of arrays. Additionally, we don't know what hardware might exist on the SoC at the periphery of our arrays. The existence of crossbars at the inputs and outputs to our arrays would allow us to permute the input and output locations between circuit mappings. For example, circuit1 might want the leftmost array input to be in0 while circuit2 wants it to be in3. An external crossbar would allow us to accommodate both circuits and decrease the area and delay of the required array, giving the user further benefits.

$$COST = 2 - .5^{(x-1)} \quad (18)$$

Thus we are presented with a need for four annealing scenarios: using a PLA with fixed I/O positions, using a PLA with variable I/O positions, using a PAL with fixed I/O positions, and using a PAL with variable I/O positions.

The differences between the annealing scenarios are shown in Figure 33. Given a PLA with fixed I/O positions, the only moves that we can make are swaps of product terms within a circuit (A). Given variable I/O positions (B), however, we can also make swaps between the inputs of a circuit or between the outputs of a circuit, which will likely provide us with further reduction of the programmable connection cost.

The outputs in a PAL put restrictions on where the product terms can be located, so the PAL with fixed I/O positions only allows product terms to be swapped within a given output OR gate (C). In the PAL where we can vary the I/O positions, we actually order the outputs by size (number of product terms) for each circuit such that the larger output gates appear at the bottom. We are then permitted to make three types of moves: swapping input positions, swapping product-term positions, and swapping output positions of equal size, as shown in (D).

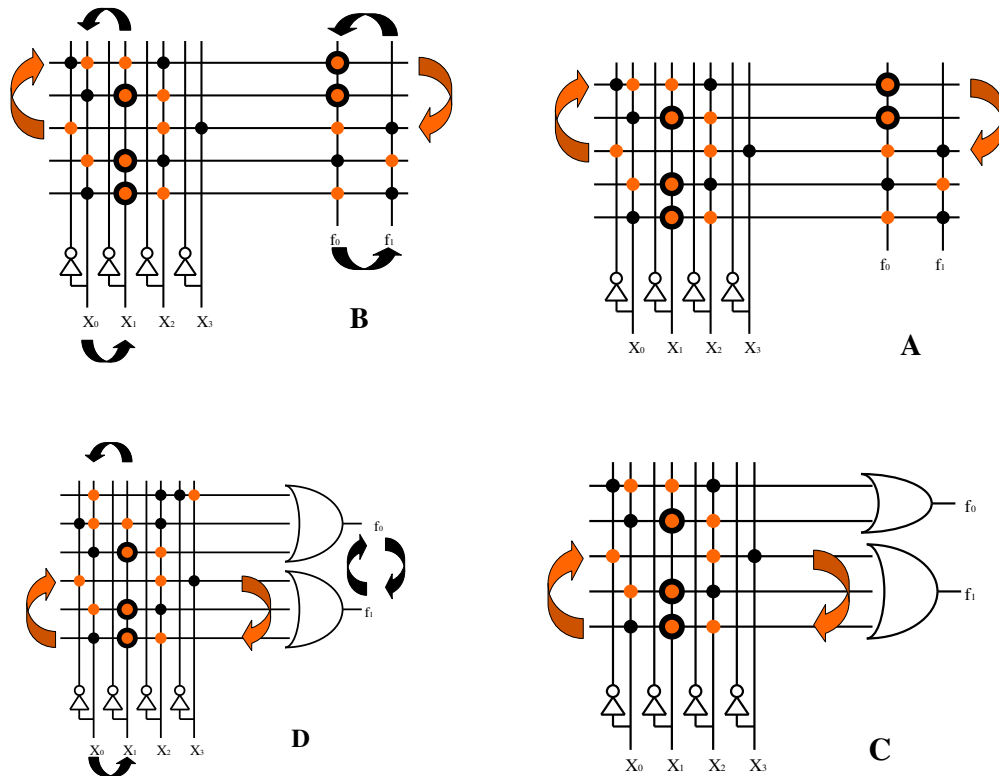


Figure 33. Allowable annealing moves for the four scenarios

For the PLA with variable I/O positions, 50% of the moves are product-term swaps and 50% are I/O swaps, with the ratio of input to output swaps equal to the ratio of inputs to outputs. For the PAL with variable I/O positions, 50% of the moves are product terms and 50% are input moves, with the output moves not currently considered because early results showed no gain from including them. The choice of performing 50% product-term and 50% I/O moves was somewhat arbitrary: we wanted to ensure that both of these

dimensions were adequately explored, especially since moves in one of these dimensions can allow new explorations in the other dimension, but providing exactly half the moves to each space was an arbitrary decision. The results that will be presented later, particularly in Table 6, show that our choice provides quality results.

When the Architecture Generator is done annealing, it creates a file that completely describes the array. This file is then read by the Layout Generator so that a layout of the array can be created. The Architecture Generator also outputs a configuration file for each circuit so that the circuit can be implemented on the created array.

6.3 Layout Generator

The Layout Generator is responsible for taking the array description created by the Architecture Generator and turning it into a full layout. It does this by combining instances of pre-made layout cells in order to make a larger design. After the cells are laid down, a compaction tool is optionally run on the design in order to create a more compact layout. The Layout Generator runs in Cadence's LayoutPlus environment, and uses a SKILL routine that was written by Shawn Phillips [19]. Designs are made in the native TSMC .18 μ process.

Figure 34 shows two PLAs that our Layout Generator created: the first PLA displays the compactness of our layouts, while the second array gives an example of the array depopulation that our algorithm achieves. Very small arrays have been shown for clarity, but the arrays we create are often orders of magnitude larger. Pre-made cells exist for every part of a PLA or PAL array, including the decoder logic needed to program the arrays. The Layout Generator simply puts together these pre-made layout pieces as specified by the Architecture Generator, thereby creating a full layout. The input file created by the Architecture Generator contains cell names and layout positions, and the SKILL routine must simply iteratively place the units as it is instructed.

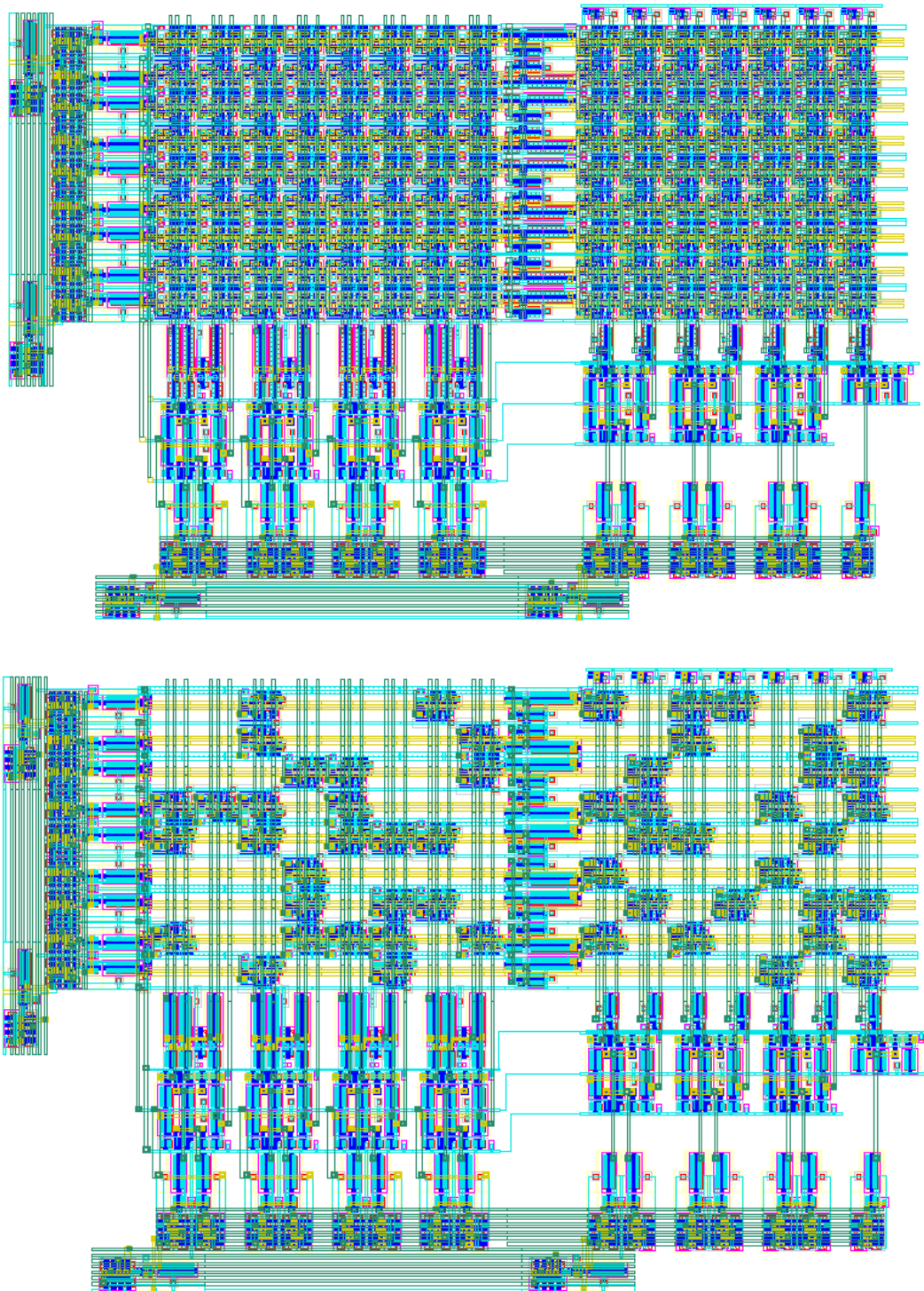


Figure 34. PLAs are created by tiling pre-made, optimized layout cells. The top PLA displays a full array, while the bottom PLA displays the effect of array depopulation

Currently, the PLAs and PALs are implemented using a pseudo-nMOS logic style (see Figure 35). PALs and PLAs are well suited to pseudo-nMOS logic because the array locations need only consist of small pull-down transistors controlled by a programmable bit, and only pull-up transistors are needed at the edges of the arrays. The programmable bits (not shown) are in series with the pull-down transistors in the arrays. The pseudo-nMOS PLAs and PALs will have small layouts, and we have found that the increased delay that comes from using this logic style is quite acceptable.

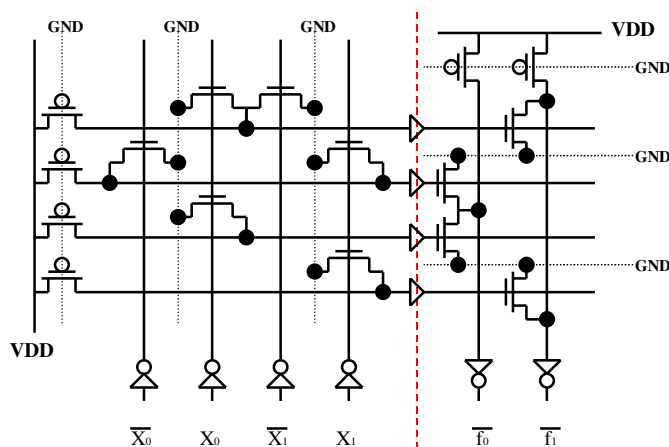


Figure 35. A PLA created using pseudo-nMOS

6.4 Methodology

The use of PALs and PLAs restricts us to the use of .pla format circuits. The first source of circuits is the ESPRESSO suite (the same circuits on which the ESPRESSO algorithm was tested). A second set of circuits comes from the benchmark suite compiled by the Logic Synthesis Workshop of 1993 (LGSynth93). As a whole, these circuits are commonly used in research on programmable logic arrays. The circuits are generally fairly small, but this suits our needs as we are currently only using single arrays to support them.

Table 5 gives information on the main circuits that we used for gathering results, including the number of inputs, outputs, product terms, and programmable connections. In sum-of-products notation, each occurrence of a variable is called a literal. For a PAL, the number of literals is equal to the number of programmable connections that are

needed in the array. For a PLA one must add the number of literals to the number of product terms in the equations in order to obtain the total number of programmable connections in the array. The connection counts in Table 5 are for PLA representations. The circuits are grouped according to size, as this will be a factor in how well our algorithms perform.

Table 5. The circuits used, with their information and groupings

Group	Circuit	Inputs	Outputs	P.Terms	Connections
1	ti	47	72	213	2573
	xparc	41	73	254	7466
2	b2	16	17	106	1941
	shift	19	16	100	493
	b10	15	11	100	1000
	table5.pla	17	15	158	2501
	misex3c.pla	14	14	197	1561
	table3.pla	14	14	175	2644
3	newcpla1	9	16	38	264
	tms	8	16	30	465
	m2	8	16	47	641
	exp	8	18	59	558
4	seq	41	35	336	6245
	apex1	45	45	206	2842
	apex3	54	50	280	3292

6.5 Results

6.5.1 Architecture Generator

The Architecture Generator uses simulated annealing to reduce the total number of programmable bits that the resultant array will require. While tools like VPR can have annealing results where costs are reduced by orders of magnitude, such large cost improvements are not possible for our annealer because our cost function is very different.

In actuality, the best cost improvement that our annealer can obtain is bounded, as shown by Figure 36. In part A we have the worst possible placement of the two circuits, and in part B we have the best possible placement. Notice that our cost only goes from 20 to 15, while the total number of actual bits we require goes from 20 to 10. These are

actually the bounds of a two circuit anneal: the cost function can never improve more than 25% and the number of programming bits required can never improve more than 50%. Similarly, with three circuits the best improvement in cost function occurs when three circuits are initially mapped to unique locations, but are all mapped onto the same locations in the final placement. For this case the maximum cost function improvement is 41.7%, while the optimal reduction in the number of programming bits is 66.7%. Similar analysis can be performed on groups of four or more circuits. Since reducing the number of bits is our final objective, the results that we present will show the number of bits required for a mapping rather than the annealing cost.

The problem of determining the minimum possible programming bit cost of a circuit mapping is very difficult. As previously mentioned, there is an $O(n^4)$ exact algorithm for determining the minimum cost of a two circuit mapping, but we have chosen not to implement the exact algorithm because we will often be dealing with more than two circuits.

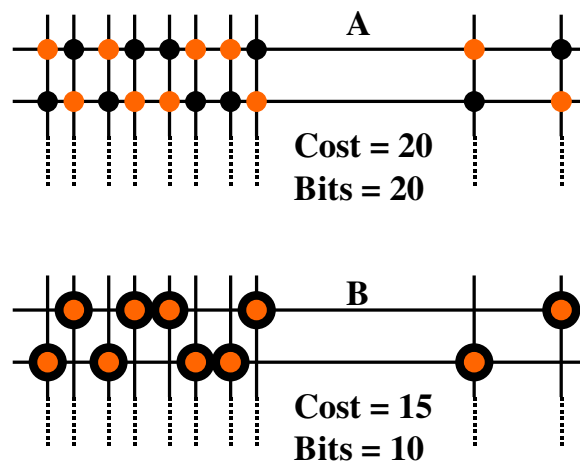


Figure 36. The best possible cost reduction for two circuits is 25%, which is 50% fewer programmable connections

Because of this, however, we do not have a method for determining the optimal bit cost of an arbitrary mapping. But we can know the optimal mapping of a circuit mapped with itself: it is simply the number of connections in the circuit, as all the connections from the first circuit should map to the same locations as the connections from the second

circuit. This can be done with any quantity of the same circuit, and the optimal solution will always remain the same. By doing this we can see how close our annealing algorithms come to an optimal mapping. Table 6 shows the results obtained from applying this self-mapping test to several circuits using each of the four algorithms: PLA-fixed, PLA-variable, PAL-fixed, and PAL-variable. The table shows that when two circuits are mapped with themselves using the PLA-fixed algorithm that the final mapping is always optimal. The PLA-variable algorithm had difficulty with only one circuit, *shift*, which was 15.21% from optimal. Note that for this example the random placement (not shown) was 92.49% worse than optimal, so our algorithm still showed major gains.

Table 6. Running the algorithms on multiple occurrences of the same circuit. The "Error" column denotes deviation from the optimal result

Alg.	Circuit	# Circuits	Optimal	Achieved	Error
PLA-Fixed	shift	2	493	493	0.00%
	table5.pla	2	2501	2501	0.00%
	newcpla1	2	264	264	0.00%
	m2	2	641	641	0.00%
	tms	2	465	465	0.00%
PLA-Var.	shift	2	493	568	15.21%
	table5.pla	2	2501	2501	0.00%
	newcpla1	2	264	264	0.00%
	m2	2	641	641	0.00%
	tms	2	465	465	0.00%
PAL-Fixed	shift	2	399	399	0.00%
	table5.pla	2	7257	7257	0.00%
	newcpla1	2	325	325	0.00%
	m2	2	2215	2215	0.00%
	tms	2	1804	1804	0.00%
PAL-Var.	shift	2	399	452	13.28%
	table5.pla	2	7257	7257	0.00%
	newcpla1	2	325	325	0.00%
	m2	2	2215	2215	0.00%
	tms	2	1804	1804	0.00%

For the PAL-fixed algorithm, all of the tests returned an optimal result. The PAL-variable algorithm had a similar result to the PLA-variable algorithm, as the *shift* circuit was only able to get 13.28% from optimal (vs. 92.23% from optimal for a random placement). The near optimal results shown in Table 6 give us confidence that our

annealing algorithms should return high quality mappings for arbitrary circuit mappings as well.

Shift was the only circuit in Table 6 that displayed suboptimal performance, so we took a closer look to see what was causing this behavior. The *shift* circuit has 19 inputs, 16 outputs, and 100 product terms, and all of product terms follow a very regular pattern (Figure 37 shows a representative portion of the circuit). The leftmost 3 input lines of the circuit are heavily populated (between 42 to 48 connections per input), while the remaining 16 input lines are all very sparsely populated, containing at most 8 connections on a single input line. The output lines are also sparsely populated, as no output line contains more than 8 connections (and most of them contain exactly 8 connections). Additionally, 98 of the 100 product terms have between 2 and 4 connections on them.

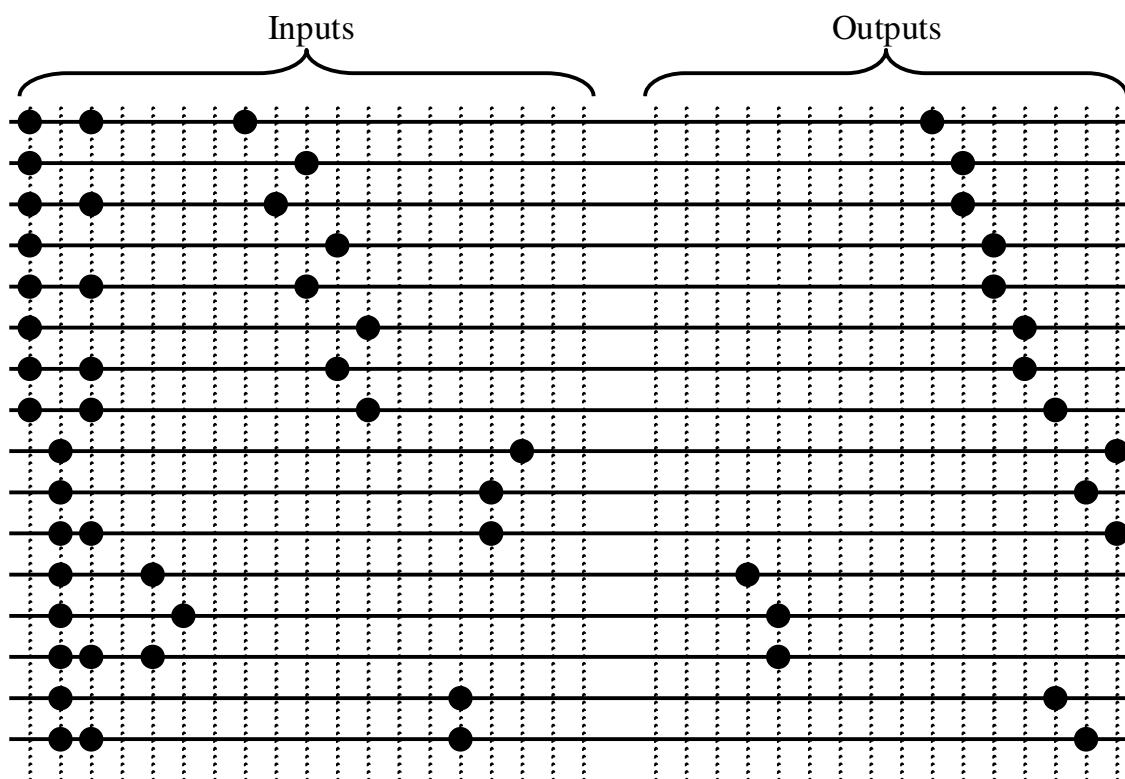


Figure 37. A representative portion of the *shift* circuit. The leftmost three input lines are heavily populated, but all other lines are very sparsely populated

For the variable algorithms, the annealer begins with circuits whose product-term, input, and output lines have been randomly permuted. In our test runs with the *shift*

circuit, the random initial state makes it nearly impossible to determine which product-term, input, or output lines from circuit1 match which product-term, input, or output lines from circuit2, because so many of these lines have an equivalent number of connections on them. The only lines that are decipherable are the first three input lines, as they have an increased number of connections on them.

I believe that the reason the *shift* algorithm performed suboptimally is that most of the product-term, input, and output lines look alike. In both of the suboptimal cases in Table 6, the second and third input lines were swapped for one of the circuits in the final mappings. Considering that the remaining input and output lines were probably quite randomly permuted and largely indecipherable from each other when this move took place, it is really no surprise that reversing the second and third input lines could have resulted in a sustainable move. Once these lines were reversed, the annealer then optimized the rest of the product-term, input, and output lines according. The resulting mapping displayed many product-term lines that were perfectly matched between the two circuits, despite the fact that the specific product-term, input, and output lines were not mapped to their perfect partners from the other circuit. I believe that the main concept to take away from this result is that, if two circuits are so similar that many of their product-term, input, or output lines look alike, then the resulting mapping may be slightly less optimal than if the perfect matchings between these lines were more clear.

Another thing to notice in Table 6 is that the PAL-fixed and PAL-variable algorithms often require many more programmable connections than the PLA algorithms on the same circuits (see the *tms* and *table5.pla* circuits). This is because, in its default state, ESPRESSO attempts to assign product terms to as many outputs as possible in order to reduce the final PLA size. This results in bad behavior for a PAL, as PALs use fixed output gates and any output sharing must be unrolled. This can cause the number of product terms in a PAL to be many times worse than in a PLA for the same circuits.

A solution to this problem would be to run ESPRESSO on each output individually and concatenate the results in order to come up with an efficient PAL representation. We did not implement this in practice, however, because we knew that we were going to be

using PLAs (rather than PALs) in our CPLD implementations, and we chose to put our efforts elsewhere. Any future work considering the elimination of programming points in a PAL should perform this step, however, to obtain the best possible PAL implementations.

Table 7 shows the results of running the PLA-fixed and PLA-variable algorithms on the different circuit groups from Table 5. The reduction in bit cost is the difference in the number of programmable connections needed between a random mapping of the circuits and a mapping performed by the specified algorithm. In the table, all possible 2-circuit mappings were run for each specific group and the results were then averaged. The same was done for all possible 3-circuit mappings, 4-circuit, etc., up to the number of circuits in the group.

Table 7. Average improvement in programming bits for PLA-Fixed and PLA-Variable algorithms over random placement as a function of circuit count

Group	# Circuits	PLA-Fixed	PLA-Var.
1	2	3.62%	14.27%
2	2	10.19%	14.52%
	3	16.26%	22.97%
	4	20.20%	28.52%
	5	23.15%	20.07%
	6	25.64%	35.46%
3	2	9.41%	16.44%
	3	14.33%	25.12%
	4	17.79%	29.81%
4	2	3.41%	19.02%
	3	6.01%	28.83%

There are some interesting things to note from the results in Table 7. Firstly, the PLA-variable algorithm always finds a better final mapping than the PLA-fixed algorithm. This is to be expected, as the permuting of inputs and outputs in the PLA-variable algorithm gives the annealer more freedom. The resulting solution space is much larger for the variable algorithm than the fixed algorithm, and it is intuitive that the annealer would find a better mapping given a larger search space. The practical implications of this are that an SoC designer will acquire better area and delay results from our reconfigurable arrays by supplying external hardware to support input and

output permutations – although the area and delay overhead of the crossbar would need to be considered to see if the overall area and delay performance is still improved.

Another thing to notice is that the reduction always increases as the number of circuits being mapped increases. This, too, is as we would expect, as adding more circuits to a mapping would increase the amount of initial disorder, while the final mapping is always close to optimally ordered. Note that this does not say that we end up with fewer connections if we have more circuits, it only says that we reduce a greater number of connections from a random mapping. The trends shown in Table 7 for the PLA algorithms hold for the PAL algorithms as well.

Another important concept is how well circuits match each other, as higher reductions will be possible when the circuits being mapped have similar sizes or a similar number of connections. With regards to array size, any circuits that are far larger than another circuit will dominate the resulting size of the PLA or PAL array, and we will be left with a large amount of space that is used by only one or few circuits, resulting in poor reduction. If the size of the resulting array is close to the sizes of each circuit being mapped to it then we would expect the array to be well utilized by all circuits. This is shown in Figure 38, which shows the bit reduction vs. array utilization. The array utilization is defined as the percentage of the final PLA's area that is being utilized by both circuits. The PLA-variable algorithm was used for these results, and the circuit pairs were chosen at random from the entire ESPRESSO and LGSynth93 benchmark suites.

Mapping circuits with a similar number of connections also results in better reductions. If circuit A has far more connections than circuit B then the total number of connections needed will be dominated by circuit A: even if we map all of the connections from circuit B onto locations used by circuit A we will see a small reduction percentage because B contained so few of the overall connections. It is intuitive that having a similar number of connections in the circuits being mapped will allow a higher percentage of the overall programmable connections to be removed. This is shown in Figure 39, where we used the PLA-variable algorithm on random circuit pairs from the

benchmark suites. A connection count of 80% means that the smaller circuit has 80% of the number of connections that the larger circuit has.

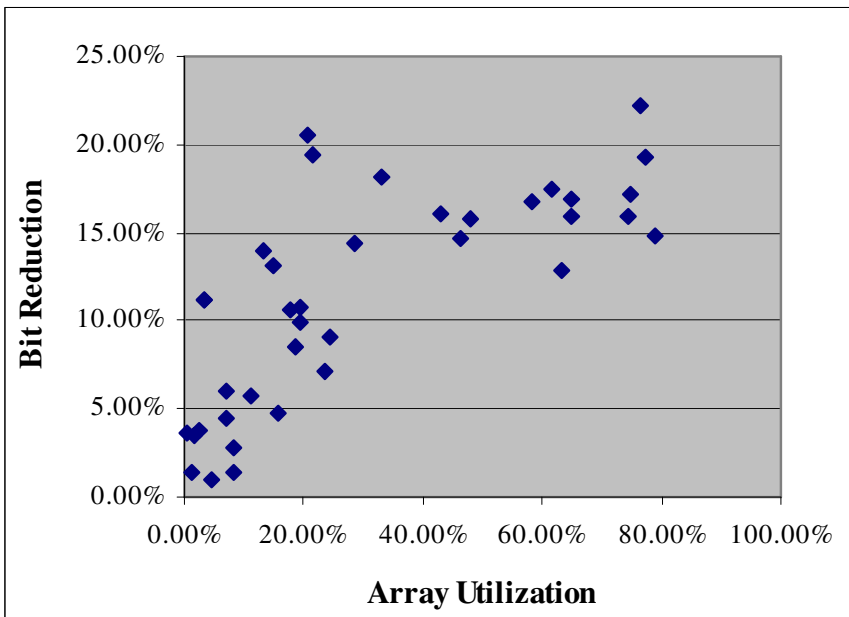


Figure 38. The bit reduction obtained vs. percent array utilization for random circuit pairs

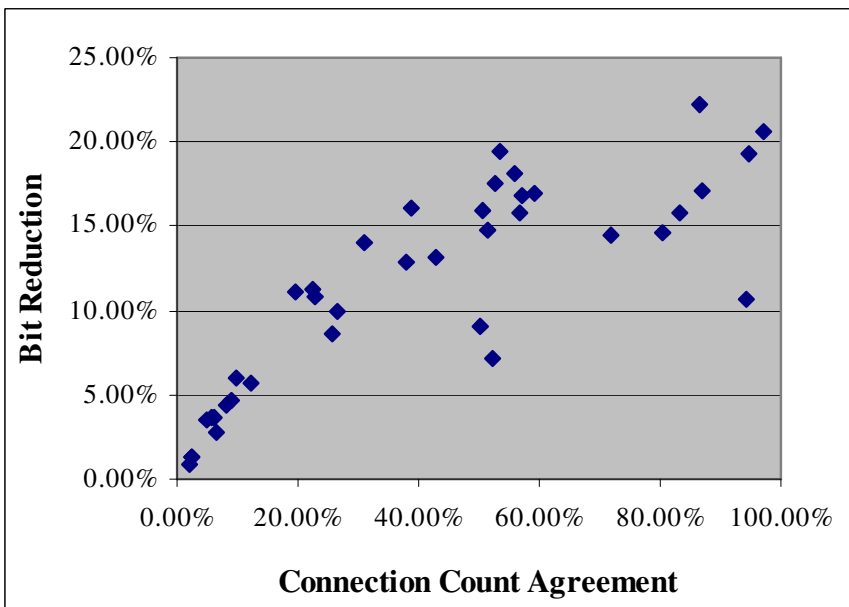


Figure 39. The bit reduction obtained vs. connection count agreement for random circuit pairs

One reason that we're trying hard to reduce the number of programmable bits is that it will allow us to use less silicon, and this will allow us to have a smaller array. The reduced array we create will be the same size but with empty space where programmable bits are not needed, and running a compactor on the layout will allow us to obtain more area savings. Additionally, the removed silicon will result in delay gains.

We used hspice to develop delay models of both the PLA and PAL arrays that we create. Table 8 shows the delay and programmable bit results obtained for several runs of the algorithms, along with average improvements over the full arrays and the random arrays (circuits randomly placed and unneeded connections removed).

All algorithms show improvements in delay over the full and random placements. Additionally, as more circuits are mapped to the array, the delay gains tend to increase.

The PLA-Variable algorithm does somewhat better than the PLA-Fixed algorithm with respect to programmable connections, but this does not scale to delay, as the PLA-V and PLA-F algorithms perform very similarly. This is because the algorithms have no concept of path criticality, and the connections that they are able to remove are often from non-critical paths. Thus, further reduction in connections does not directly lead to further reduction in delay.

Table 8. Reductions obtained in number of programmable bits and delay for PLA/PAL algorithms

Circuits	PLA Algorithms								PAL Algorithms							
	Programmable Bits				Delay (ps)				Programmable Bits				Delay (ps)			
	Full	Rand.	PLA-F	PLA-V	Full	Rand.	PLA-F	PLA-V	Full	Rand.	PAL-F	PAL-V	Full	Rand.	PAL-F	PAL-V
misex3c.pla, table3.pla	8274	3675	3165	2998	3620	3089	2901	2905	19936	8611	8155	7739	8505	7218	6846	6760
alu2, f51m	2156	683	557	538	1708	1081	916	908	2220	593	544	486	1627	969	914	866
ti, xparc	42418	9796	9452	8207	5343	4578	4561	4512	269686	55957	55548	51610	33696	25908	25715	25409
b2, shift, b10	5830	2890	2510	2270	2329	2065	1999	1937	37620	10150	9627	9098	10557	6798	6581	6544
newcpla1, tms, m2	1598	1000	862	779	1268	1236	1216	1208	6984	3437	2998	2527	3993	3906	3831	3278
gary, b10, in2, dist	6664	3458	2658	2026	2760	2658	2610	2569	15010	5710	4852	3298	4800	3715	3392	3190
newcpla1, tms, m2, exp	2124	1319	1072	956	1459	1420	1323	1372	7218	3954	3276	2679	4095	4005	3940	3406
gary, shift, in2, b2, dist	7480	4410	3516	2960	2785	2727	2646	2615	39216	12941	11437	9751	10887	7794	7289	6741
b2, shift, b10, table5.pla, misex3c.pla, table3.pla	18321	6600	4914	4521	4015	3718	3009	3118	92796	21081	16987	13453	15692	10847	9514	8513
Average Improvement Over Full	-	53.3%	61.4%	65.8%	-	10.6%	16.0%	16.1%	-	65.0%	68.9%	73.5%	-	22.3%	26.0%	31.3%
Average Improvement Over Random	-	-	16.7%	25.9%	-	-	6.4%	6.5%	-	-	10.6%	23.1%	-	-	5.1%	11.4%

The PAL-Variable algorithm performs better than the PAL-Fixed algorithm in terms of both programmable connections and delay. This is largely because the PAL-V algorithm is able to permute the outputs (and therefore the output OR-gates), resulting in smaller OR-gates in many circumstances.

On average, the PLA-Fixed and PLA-Variable algorithms improved upon the delay of a full PLA array by 16.0% and 16.1% respectively. The PAL-Fixed and PAL-Variable algorithms improved upon the delay of a full PAL array by 26.0% and 31.3% respectively. Overall, delay improvements of 5.0% to 11.4% were achieved vs. a random placement.

6.5.2 *Layout Generator*

In the Architecture Generator, unneeded programmable connections are removed from the PLA and PAL arrays that we create. This leaves the arrays full of randomly distributed empty space that a compaction tool should be able to leverage in order to make a smaller, more compact layout.

We took several PLA and PAL layouts and applied Cadence's compactor to them, but found that the compactor was unable to reduce the area of any of the arrays (and in fact resulted in a larger area implementation in all cases). Applying the compactor to the depopulated PLA from Figure 34 resulted in the layout shown in Figure 40.

The failure of the compactor is due to the high regularity of PLA and PAL arrays. The compactor iteratively attempts to compact in the vertical and horizontal directions, but PLAs and PALs have strong vertical and horizontal relationships between array elements which prevent the compactor from making any headway.

One interesting note is that the compacted image in Figure 40 actually does not conform to all of the design rules for the layout process. We were unable to get Cadence's compactor to fully adhere to the design rules while compacting, so the resulting compacted PLAs are probably a lower bound for what a legal compaction could have achieved.

6.6 Conclusions

This chapter presented our work in automating the creation of domain-specific PLAs and PALs, including an Architecture Generator that efficiently maps circuits to a PLA or PAL array and a Layout Generator which tiles pre-made layouts in order to create

a full VLSI layout of the PLA or PAL array. Delay improvements of 16% to 31% were achieved over full arrays, but compaction was unable to provide us with any area improvements. The largest improvements were obtained when the PLA/PAL inputs and outputs were permutable, but an SoC designer would need to examine the overhead of input and output crossbars to and from the arrays in order to decide whether an area or delay gain would actually be achieved.

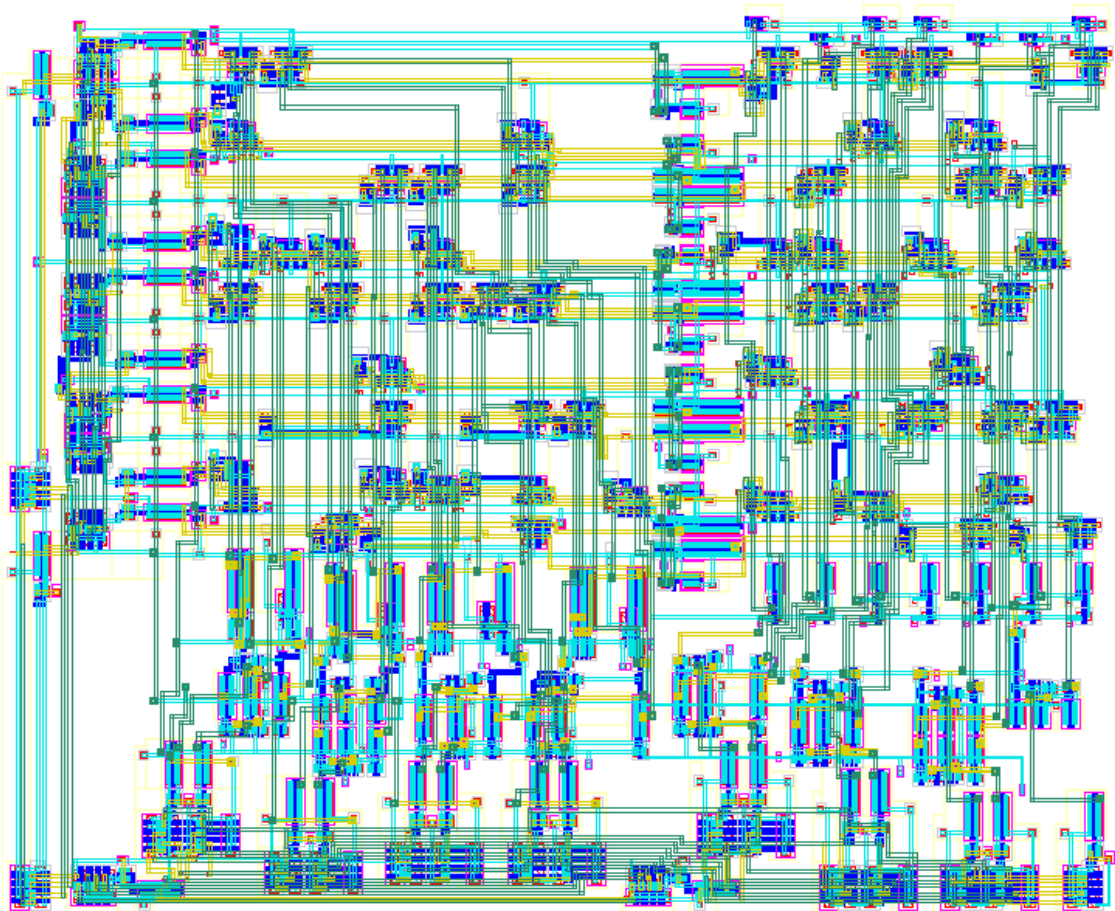


Figure 40. The result of applying a compactor to the depopulated PLA from Figure 34. This layout is actually 5% larger than the uncompact layout in Figure 34

If the PLA or PAL array being created will only be used to implement the circuits that were used to design the array, then depopulating the arrays is a good idea, as it will provide delay gains over a fully populated array. If other designs are going to be implemented on the array, however, the removal of programmable connections will make

it unlikely that a future circuit would successfully map to the array. Thus the removal of programmable connections from a PLA or PAL is not suggested if unknown circuits are going to be mapped to the array in the future.

Generally speaking, realistic PLA and PAL arrays are limited in size due to area and delay considerations, and cannot support very large circuits. When performing large amounts of computation, better performance can be achieved through the use of architectures that utilize a large number of smaller functional units. These considerations make standalone PLA or PAL arrays unattractive as reconfigurable solutions for SoC devices.

7 Logic in Domain-Specific CPLDs

In the previous chapter we examined the methods in which a stand-alone PLA or PAL can be tailored to an application domain. This chapter will build upon that work by combining multiple of these units into a CPLD structure. The focus of this chapter is on the CPLD logic elements, so in order to isolate the logic from routing considerations/restrictions we will be connecting the elements through a full crossbar. In the next chapter we will then focus on ways to modify the CPLD's interconnect structure.

When progressing to CPLDs, we chose to make a decision between using either PLAs or PALs as the logic element for the architectures. The majority of the tech-mapping algorithms available for product-term architectures map to PLA arrays rather than PAL arrays. This includes an algorithm called PLAMap (see chapter 2), which is currently the best academic tech-mapping algorithm for PLA-based CPLDs. While it would be possible to modify one of these PLA-based algorithms to map to PAL arrays, it is unclear whether such an undertaking would result in architectures that are better than what we can get using PLAs, so we determined that our time would be better spent exploring PLA-based CPLD architectures using existing tech-mapping tools. The decision to use PLAs has some useful side effects as well, including a smaller design space to explore (because PALs must have their output OR-gate sizings specified), and greater flexibility due to fully programmable OR-planes. For the rest of this dissertation we will only be considering PLAs as the logic units in CPLDs.

In the previous chapter, the majority of our effort was put into removing unneeded programmable connections in the arrays – a process which we ultimately concluded to be too detrimental to the flexibility of the stand-alone devices. This is also true when we try to combine multiple PLAs into a CPLD: reducing the connectivity in the logic elements will make it difficult or impossible to map future designs to the architecture. Because of

this, removing programmable connections from the PLA arrays is no longer considered. This chapter details the algorithms used to tailor a CPLD to an application domain by varying the input capacity, product-term capacity, and output capacity of the PLAs within the CPLD architecture. This represents a complete flow (from SoC designer through our automated tools and back to the designer) for full-crossbar-based CPLDs.

7.1 Tool Flow

Figure 41 shows the tool flow used for tailoring CPLD logic to a particular application domain. The input from the SoC designer is a specification of the target domain, containing a set of circuits that the architecture must support. These circuits are fed into an Architecture Generator, which will find a CPLD architecture that provides good performance for the selected domain. The architecture description outputted by the Architecture Generator is then sent to a Layout Generator, which creates a full VLSI layout of the specified CPLD architecture in the TSMC .18 μ process.

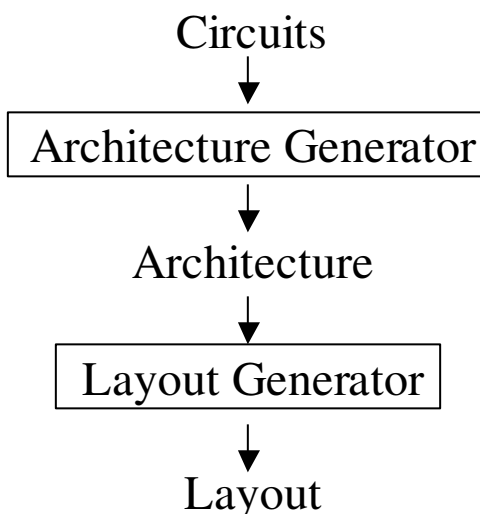


Figure 41. Totem-CPLD Tool Flow

7.2 Architecture Generator

The Architecture Generator is responsible for reading in multiple circuits and finding a CPLD architecture that supports the circuits efficiently. Search algorithms are used to

make calls to PLAmapping, after which the results are analyzed according to area and delay models that we have developed. The algorithms then make a decision to either make further calls to PLAmapping, or to exit and use the best CPLD architecture that has been found. This is shown graphically in Figure 42. PLAmapping assumes full connectivity between the PLAs, and the Architecture Generator accommodates this by connecting all the PLAs through a full crossbar.

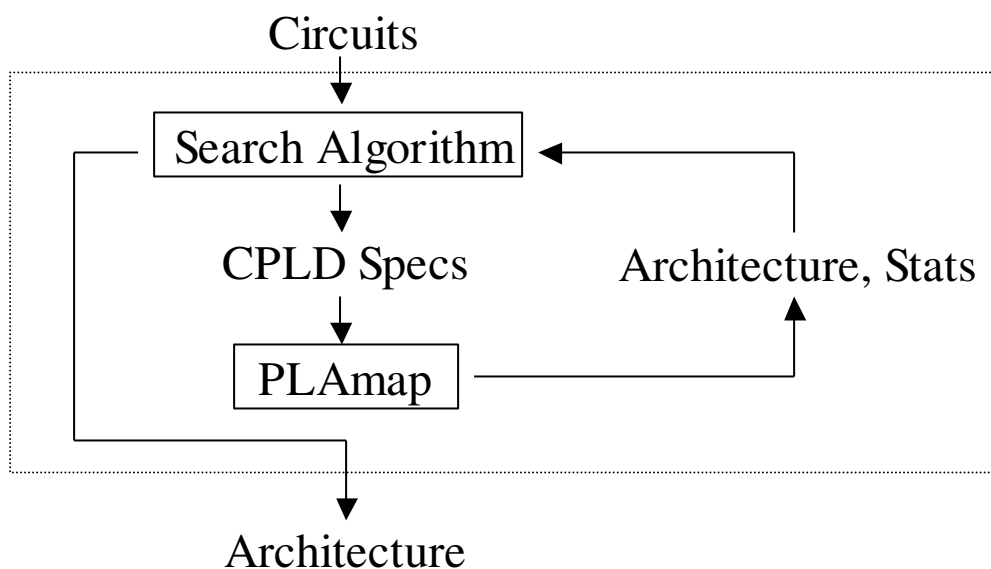


Figure 42. Architecture Generator

The Architecture Generator is responsible for finding a PLA size that leads to an efficient CPLD architecture for the given domain. PLAs are specified by their number of inputs (IN), product terms (PT), and outputs (OUT), so the search space for the Architecture Generator is three-dimensional. Searching the entire 3-D space is not viable, as calls to PLAmapping can take on the order of hours for larger circuits, and our ultimate goal is to find a suitable CPLD architecture in a matter of hours or days. Also, for each PLA architecture on which we test a domain, PLAmapping must be called once for each circuit in the domain. Clearly, minimizing the number of PLAmapping calls is important to our runtime. Effective algorithms such as simulated annealing and particle swarm require too much time for our scenario, and smart algorithms will be required if we wish to acquire good results in the 3-D space using relatively few data points.

In order to gain some intuition about the search space, we ran five random LGSynth93 circuits through PLAMap and acquired a coarse representation of the 3-D space for each circuit. We tested PLAs with input (IN) values of 4, 8, 12, 16, 20, 24, and 28, product term (PT) values of $.5*IN$, $1*IN$, $2*IN$, $3*IN$, and $4*IN$, and output (OUT) values of $.25*IN$, $.5*IN$, $.75*IN$, and $1*IN$. All possible permutations of these values were tested, so a total of 140 PLA sizes were tested for each circuit. The area-delay product for each test case was acquired, and the geometric mean was calculated for each PLA size across the five test circuits.

We chose to determine the ranges of PLA variables that perform well, rather than simply find the best single architecture across the five test circuits. This is because we did not want to rely too heavily on the test circuits we were using, as they might not be very representative of the overall set of circuits that we would eventually use. In order to do this, we took the geometric mean of all of the architectures with $IN=4$, for all architectures with $IN=8$, and so on for each of the three PLA variables. The results of this are shown in Table 9.

Table 9. Area-Delay results for different PLA parameters in our test circuits

IN	Geo. Mean	PT	Geo. Mean	OUT	Geo. Mean
4	23.63	$.5*IN$	17.91	$.25*IN$	3.83
8	3.85	$1*IN$	5.10	$.5*IN$	4.47
12	2.58	$2*IN$	3.49	$.75*IN$	6.40
16	3.97	$3*IN$	3.69	$1*IN$	8.42
20	4.13	$4*IN$	4.32		
24	5.62				
28	7.15				

As Table 9 displays, the best results were obtained for IN values of 8 to 16, for PT values of $2*IN$ to $4*IN$, and for OUT values of $.25*IN$ to $.75*IN$. In order to hone in on these regions, we next discounted all of the PLA architectures that used variables outside of these ranges: so we were now only considering PLAs with IN values of 8, 12, or 16, PT values of $2*IN$, $3*IN$, or $4*IN$, and OUT values of $.25*IN$, $.50*IN$, or $.75*IN$. The results for these ranges are shown in Table 10.

As shown in Table 10, the best two IN values for our test circuits are 12 and 8, the best two PT values are $3*IN$ and $2*IN$, and the best two OUT values are $.5*IN$ and

.25*IN. This has provided us with a region of PLA sizes that are effective, as well as a relationship between the IN, PT, and OUT variables that tends to provide quality results. To generalize, a ratio of 1x to 2x to .5x for the IN, PT, and OUT variables respectively was found to consistently provide good results. Within the scope of these ratios, CPLDs with roughly 10-20-5 PLAs were very effective. These results will be leveraged in the formulation of our architecture generation algorithms.

Table 10. Area-Delay results for different PLA parameters in our test circuits. Data includes only PLAs with the PLA parameters shown in the table

IN	Geo. Mean	PT	Geo. Mean	OUT	Geo. Mean
8	2.30	2*IN	1.88	.25*IN	1.84
12	1.35	3*IN	1.78	.5*IN	1.74
16	2.35	4*IN	2.18	.75*IN	2.29

Another thing that we observed from these preliminary results is that results tend to get better as you approach the optimal point, and worse as you move away from it. This observation led us to the concept of breaking the 3-D space into three 1-D spaces, which can be searched sequentially and in much less time. Specifically, our algorithms will start by searching for a good input size (while keeping a 1x-2x-.5x IN-PT-OUT relationship), next search for a good output size, and finish by searching for a good product-term size.

Displaying the behavior of the 3-D space in a direct fashion would require four dimensions, so we will do it by displaying the slices of the 3-D space that we see in our algorithms. For our “large” sample domain, Figure 43 displays the 1-D space explored by an input step, Figure 44 displays the 1-D space explored by an output step, and Figure 45 displays the 1-D space explored by a product-term step. Notice that all three of the graphs are relatively concave, and that results do tend to get better as you approach the optimal point in a 1-D slice of the 3-D space. But these 1-D slices (and therefore the 3-D space) are far from being perfectly behaved, as there are many small perturbations in the smoothness that lead to local optima. Using our method of sequential 1-D searches will provide quality results despite the presence of these local optima, but we need to ensure that our algorithms are robust enough that they can avoid getting trapped in the suboptimal regions in these graphs.

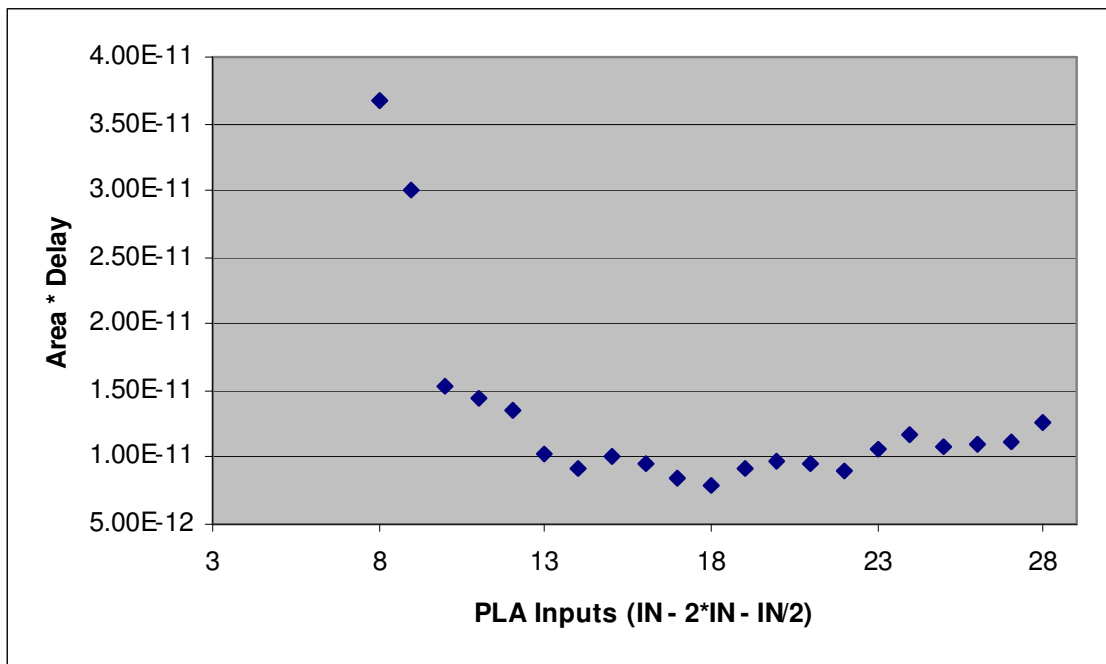


Figure 43. Results of a 1-D search through the PLA space for the “large” sample domain, using a fixed ratio of 1x-2x-.5x for the IN-PT-OUT values

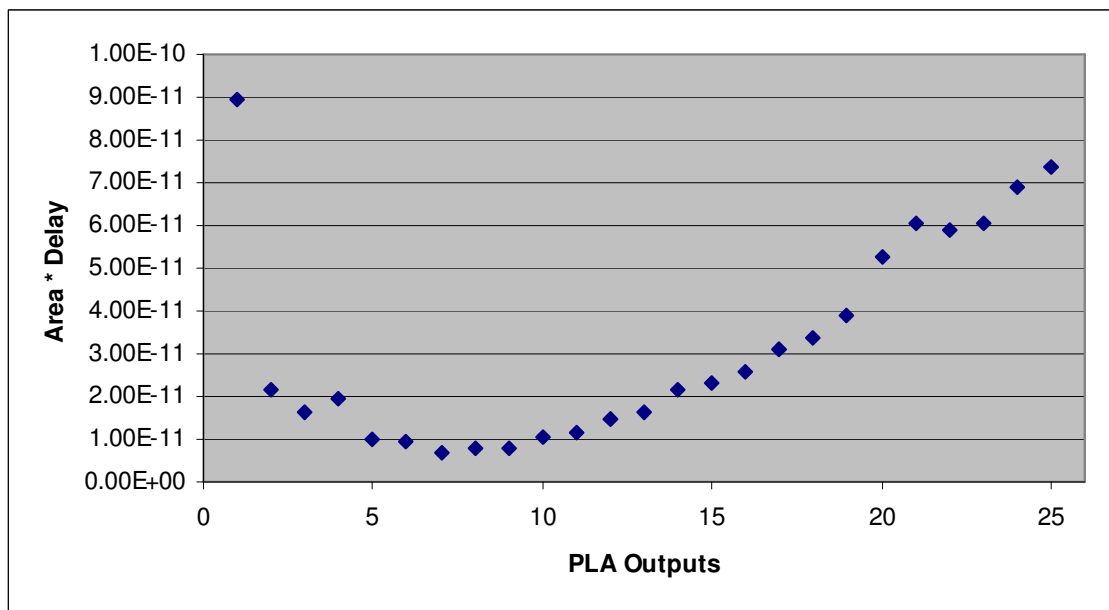


Figure 44. Results of a 1-D search through the PLA space for the “large” sample domain, starting from the best point in Figure 43 (18-36-9) and varying only the output value

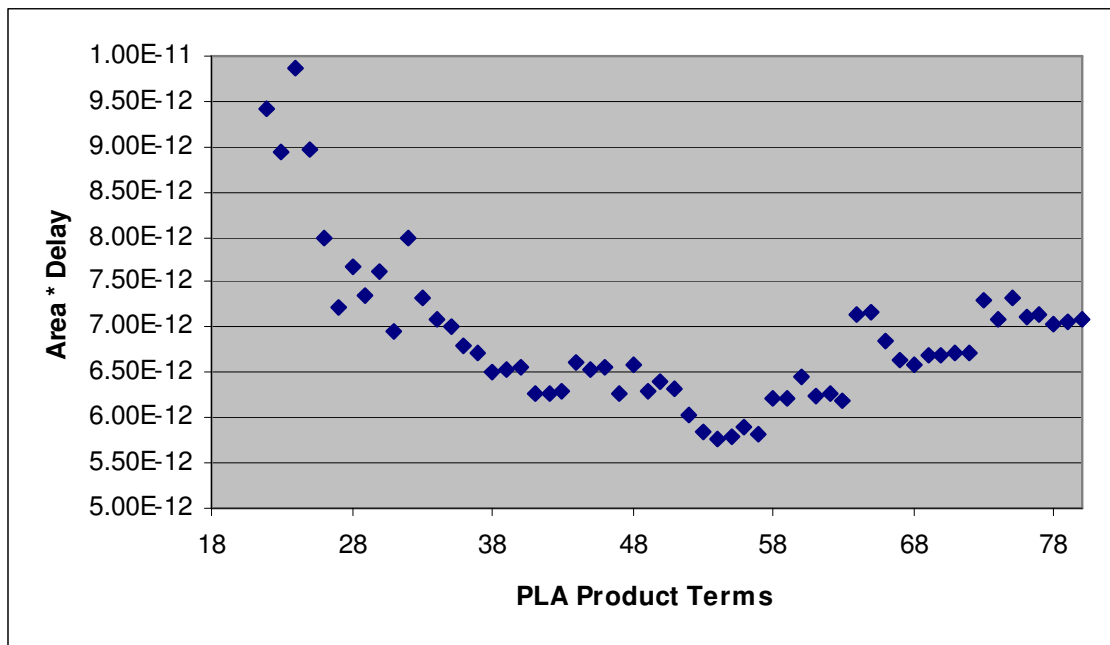


Figure 45. Results of a 1-D search through the PLA space for the "large" sample domain, starting from the best point in Figure 44 (18-36-7) and varying only the product term value

Our architectures are evaluated using the metric of area-delay product. When reported for a domain, the area-delay product consists of the worst-case area implementation in the domain (since the reconfigurable CPLD must be large enough to hold each of the circuits), multiplied by the average case delay of the domain. The area model for this calculation is derived from the actual sizings of the VLSI layout components that we created, and the delay model was acquired by creating circuits in layoutPlus and simulating them with hspice in order to acquire their worst-case delay characteristics. This is described in more detail in Chapter 5.

7.2.1 Search Algorithms

We developed four different search algorithms with the aim of finding good CPLD architectures: Hill Descent, Successive Refinement, Choose N Regions, and Run M Points. All algorithms break the 3-D search space into 1-D steps by searching for good input, output, and product-term sizes, in that order. Additionally, the input step always uses PLAs with a 1x-2x-.5x IN-PT-OUT ratio, while the output and product-term steps

always alter ONLY the output and product-term values (respectively) from data point to data point. At a high level, each of the basic search algorithms looks the same: the pseudocode for the general search algorithm is shown in Figure 46. Note that we will discuss some algorithm add-ons later in this section that will add some complexity to the basic description discussed here.

```

generalSearchAlgorithm()
{
    input circuits;      //circuits in the domain
    output in, pt, out; //the PLA values for the preferred CPLD arch.

    {in, pt, out} = runInputSearchGeneral(circuits);

    {in, pt, out} = runOutputSearchGeneral(circuits, in, pt, out);

    {in, pt, out} = runPTermSearchGeneral(circuits, in, pt, out);

    return {in, pt, out};
}

```

Figure 46. General pseudocode for the search algorithms

7.2.1.1 Hill Descent

The Hill Descent algorithm is the first algorithm that we developed, and the most basic. Like all of our algorithms, the 3-dimensional search space is broken into three sequential 1-dimensional searches. The input search step starts by running PLAMap on architectures with 10-20-5 and 12-24-6 PLAs. Whichever result is better, we continue to take results in that direction (i.e. smaller or larger PLAs), keeping the 1x-2x-.5x ratio intact and performing steps of IN = +/-2. We continue until a local optimum is reached, as determined by the first result that does not improve upon the last result. At this point we explore the PLAs with IN = +/-1 of the current local optimum. The best result is noted, and the input value is permanently locked at this value, thus ending the input step. This is shown graphically in Figure 47, and the pseudocode for it is shown in Figure 48.

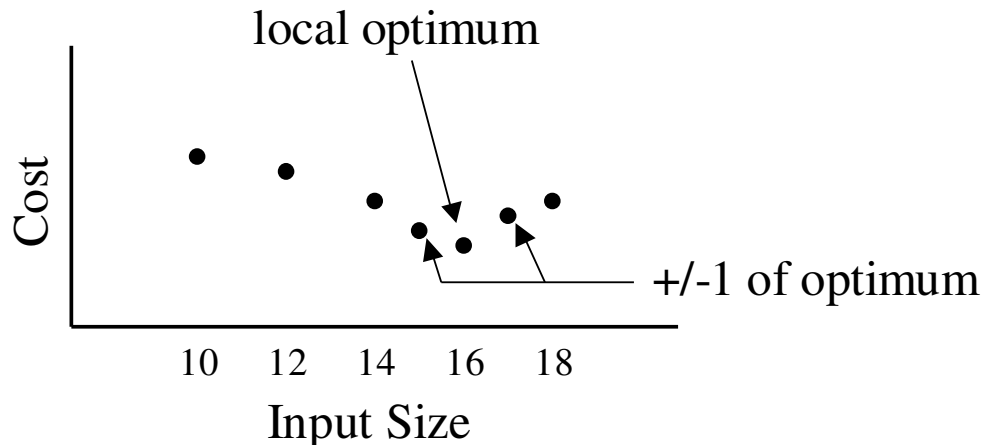


Figure 47. Hill Descent Algorithm

The output optimization step occurs next. The first data point in this step is the local optimum from the input step, and the second data point is acquired by running PLAmapping on a PLA with one more output than the current optimum (IN and PT do not change). Again, we descend the hill by altering OUT by ± 1 until the first result that does not improve upon the previous result. At this point we lock the output value and proceed to the product-term optimization step. The product-term optimization step repeats the process from the previous two steps, varying the PT value by ± 2 until the descent stops. At this point, the PT values ± 1 of the optimum are taken, and the best overall result seen is the output of the algorithm. The output and product-term steps are shown in pseudocode in Figure 49 and Figure 50 respectively.

The Hill Descent algorithm is decidedly greedy, as it always moves in the direction of initial improvement. It also has no method for avoiding local minima, as any minimum will stop the current step. Therefore it is somewhat difficult for this algorithm to find architectures that vary much in size from the 10-20-5 PLA starting point, but decent results are still obtained due to the fact that the 10-20-5 starting point is a relatively good point in the 3-D search space.


```

runInputSearchHD(circuits)
{
    output in, pt, out; //variables for the PLA-size
    result1, result2; //objects hold the results of running PLAmapping
    gettingBigger; //whether our PLAs are getting bigger or smaller

    result1 = runPLAmapping(circuits, 10, 20, 5);
    result2 = runPLAmapping(circuits, 12, 24, 6);
    //Determine whether to search larger or smaller PLA sizes
    if(result2 < result1) {
        gettingBigger = true;
        in = 14; pt = 28; out = 7;
    } else {
        gettingBigger = false;
        in = 8; pt = 16; out = 4;
        result2 = result1; //keep best result in result2
    }
    //Continue to go until we stop improving
    while((result1 = runPLAmapping(circuits, in, pt, out)) < result2) {
        result2 = result1;
        if(gettingBigger) {
            in += 2;
        } else {
            in -= 2;
        }
        pt = 2*in; out = .5*in;
    }
    //We stopped improving, so go back to the best data point
    if(gettingBigger) {
        in -= 2;
    } else {
        in += 2;
    }
    //Run the points next to our best point
    if(result1=runPLAmapping(circuits, in+1, 2*(in+1), .5*(in+1)) < result2)
        result2 = result1;
    if(result1=runPLAmapping(circuits, in-1, 2*(in-1), .5*(in-1)) < result2)
        result2 = result1;

    //return the PLA variables of the best result
    return {result2.in(), result2.pt(), result2.out()};
}

```

Figure 48. Pseudocode for the input step of the Hill Descent algorithm

```

runOutputSearchHD(circuits, in, pt, out)
{
    result1, result2; //objects hold the results of running PLAmapping
    gettingBigger;    //whether our PLAs are getting bigger or smaller

    result1 = runPLAmapping(circuits, in, pt, out);
    result2 = runPLAmapping(circuits, in, pt, (out+1));
    //Determine whether to search larger or smaller PLA sizes
    if(result2 < result1) {
        gettingBigger = true;
        out +=2;
    } else {
        gettingBigger = false;
        out -= 1;
        result2 = result1; //keep best result in result2
    }
    //Continue to go until we stop improving
    while((result1 = runPLAmapping(circuits, in, pt, out)) < result2) {
        result2 = result1;
        if(gettingBigger) {
            out += 1;
        } else {
            out -= 1;
        }
    }
    //We stopped improving, so return PLA variables of best result
    return {result2.in(), result2.pt(), result2.out()};
}

```

Figure 49. Pseudocode for the output step of the Hill Descent algorithm

7.2.1.2 Successive Refinement

The successive refinement algorithm is intended to slowly disregard the most unsuitable PLA architectures, thereby ultimately deciding upon a good architecture by process of elimination. In the input optimization step (Figure 51), data points are initially taken for PLAs with input counts ranging from 4 (lower bound) to 28 (upper bound) with a step size of 8. So initially, 4-8-2, 12-24-6, 20-40-10, and 28-56-14 PLAs are run (part a in Figure 51). The left and right edges are then examined, and regions that are unlikely to

```

runPTermSearchHD(circuits, in, pt, out)
{
    result1, result2; //objects hold the results of running PLAmapping
    gettingBigger; //whether our PLAs are getting bigger or smaller

    result1 = runPLAmapping(circuits, in, pt, out);
    result2 = runPLAmapping(circuits, in, pt+2, out);
    //Determine whether to search larger or smaller PLA sizes
    if(result2 < result1) {
        gettingBigger = true;
        pt += 4;
    } else {
        gettingBigger = false;
        pt -= 2;
        result2 = result1; //keep best result in result2
    }
    //Continue to go until we stop improving
    while((result1 = runPLAmapping(circuits, in, pt, out)) < result2) {
        result2 = result1;
        if(gettingBigger) {
            pt += 2;
        } else {
            pt -= 2;
        }
    }
    //We stopped improving, so go back to the best data point
    if(gettingBigger) {
        pt -= 2;
    } else {
        pt += 2;
    }
    //Run the points next to our best point
    if(result1=runPLAmapping(circuits, in, pt+1, out) < result2)
        result2 = result1;
    if(result1=runPLAmapping(circuits, in, pt-1, out) < result2)
        result2 = result1;

    //return the PLA variables of the best result
    return {result2.in(), result2.pt(), result2.out()};
}

```

Figure 50. Pseudocode for the product-term step of the Hill Descent algorithm

provide good results are trimmed (shaded regions in part a). The step size is then halved, and the above process is repeated (part b). This occurs until we have performed an exploration with a step size of 1 (part d). The pseudocode for this input step is shown in Figure 53.

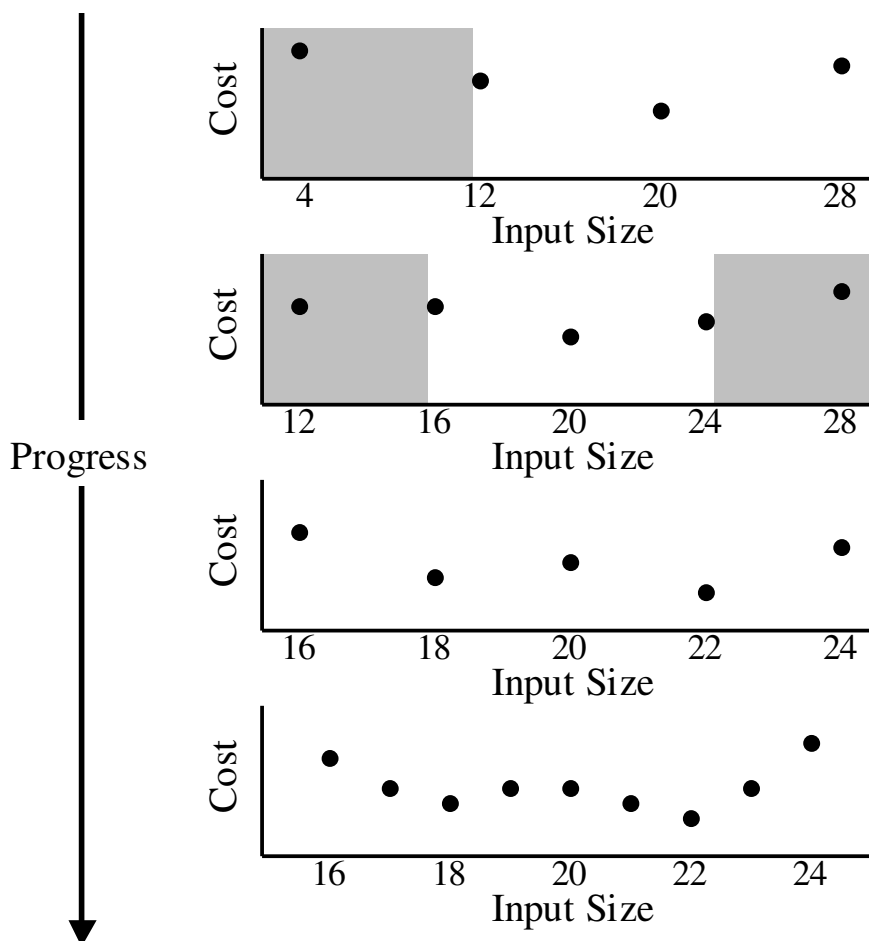


Figure 51. Input optimization step of the Successive Refinement algorithm. At each iteration, shaded regions are trimmed (not including the point at their edge) and the step size halved

The trimming algorithm deserves some additional consideration before we introduce the pseudocode. Because our 3-D PLA search space is generally well behaved, we expect our 1-D searches to come up with graphs that have only one minimum, with data progressively getting worse as we move away from the minimum. The left half of the graph in Figure 52 displays points that are well behaved. For well behaved 1-D sections, the trimming algorithm simply trims away points until it reaches three points that form a

valley, representing a minimum. These three points are not trimmed. This is shown in the Figure 52.

Because our space isn't always well behaved, however, we expect some regions of the 1-D space to have local minima. We always try to keep minima away from the edges of our current intervals by trimming only up to any three points that create a valley, ensuring that the edge is not a local minimum. Thus, if a local minimum appears at the very edge of our current interval of consideration, it is because a new data point has been explored causes the edge point to be a minimum. An example of this is shown on the right edge of Figure 52, where the second point from the right has just been explored. In these instances, the local optimum at the interval edge gets trimmed, as shown. The only exception to this is if the point at the interval edge is the current global minimum, in which case we do not trim it. The pseudocode for the trimming algorithm is shown in Figure 54, including how to deal with special cases such as ties.

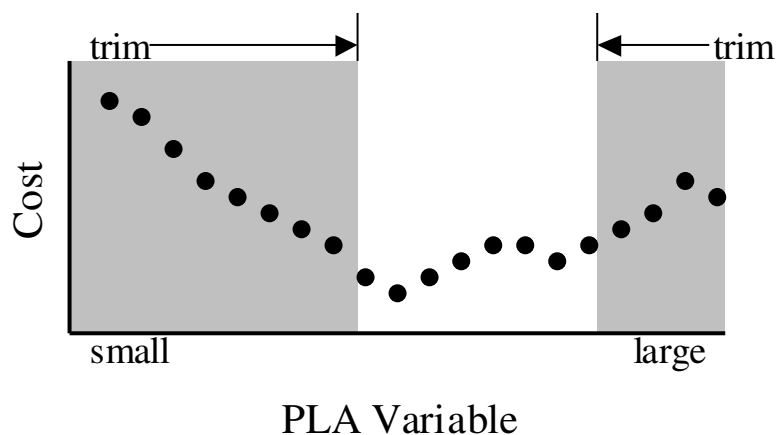


Figure 52. Trimming occurs from the edges until reaching three points that make a valley. Local optima that appear at interval edges are trimmed unless they are the current global minimum

For the output optimization step, the IN and PT values are locked at the best result we found in the input step. The output values are now varied according to the above refinement algorithm, using an initial lower bound of 1, upper bound of 25, and step size of 8. The recursion again continues until the results for a step size of 1 have been taken, at which point we lock the IN and OUT values. The product-term optimization step next repeats this process for PT values between 2 and 90, after which the best result is

returned as the best architecture found. The pseudocode for the output and product-termssteps is shown in Figure 55 and Figure 56 respectively.

```

runInputSearchSR(circuits)
{
    output in, pt, out; //variables for the PLA-size

    leftEdge = 4;      //variables holding the in value of the left
    rightEdge = 28;   //and right edges of our search window
    stepSize = 8;     //current stepSize for taking data
    results[];        //holds my result objects
    bestResult;       //holds the best result at the end

    //While we haven't completed taking data at a step size of 1
    while(stepSize > 0) {

        //For the interval and step size, add the results we don't have
        for(i=leftEdge; i<=rightEdge; i+= stepSize) {
            results = {results, runPLAmap(circuits, i, 2*i, .5*i)};
        }
        //Trim regions of poor results from the left and right edges
        leftEdge = trimFromLeftIN(leftEdge, rightEdge, stepSize, results);
        rightEdge = trimFromRightIN(leftEdge, rightEdge, stepSize, results);
        //halve the step size
        stepSize = floor(stepSize/2);
    }

    //get the best result within the interval
    bestResult = getBestResultInInterval(leftEdge, rightEdge, results);
    //return the PLA variables of the best result
    return {bestResult.in(), bestResult.pt(), bestResult.out()};
}

```

Figure 53. Pseudocode for the input step of the Successive Refinement algorithm

```

trimFromLeftIN(leftEdge, rightEdge, stepSize, results)
{
    newLeftEdge = leftEdge; //the new left edge we return
    bestResult = results.getBestResult(); //the best result we've seen

    //sort the results such that the smallest PLA is result[0]
    //we are assuming here that result[0] is the point at leftEdge
    results.sortBySize();

    i = 0; //used for walking through results

    //continue trimming until a point resists trimming, then break
    while(1) {
        //if the left point is better than the second point (rare case),
        //we trim it unless it is our best current point
        if(result[i] < result[i+1]) {
            if(result[i] == bestResult) {
                break;
            } else {
                newLeftEdge = result[i+1].in();
            }
        }
        //if the left point is worse than the second point (typical case),
        //trim it unless the three points create a valley
        else {
            if(result[i+1] < result[i+2]) {
                break;
            } else {
                newLeftEdge = result[i+1].in();
            }
        }
        i++; //increment i to consider next point for trimming
    }
    return newLeftEdge; //return the new left edge
}

```

Figure 54. Pseudocode for trimming the left edges of an interval in the input step of the Successive Refinement algorithm. The "trim from right" case is symmetrical to this. For the output and product-term steps, simply replace all ".in()" with ".out()" or ".pt()" respectively

```

runOutputSearchSR(circuits, in, pt, out)
{
    leftEdge = 1;    //variables holding the in value of the left
    rightEdge = 25; //and right edges of our search window
    stepSize = 8;   //current stepSize for taking data
    results[];     //holds my result objects
    bestResult;    //holds the best result at the end

    //While we haven't completed taking data at a step size of 1
    while(stepSize > 0) {

        //For the interval and step size, add the results we don't have
        for(i=leftEdge; i<=rightEdge; i+= stepSize) {
            results = {results, runPLAmap(circuits, in, pt, i)};
        }
        //Trim regions of poor results from the left and right edges
        leftEdge = trimFromLeftOUT(leftEdge, rightEdge, stepSize, results);
        rightEdge = trimFromRightOUT(leftEdge, rightEdge, stepSize, results);
        //halve the step size
        stepSize = floor(stepSize/2);
    }

    //get the best result within the interval
    bestResult = getBestResultInInterval(leftEdge, rightEdge, results);
    //return the PLA variables of the best result
    return {bestResult.in(), bestResult.pt(), bestResult.out()};
}

```

Figure 55. Pseudocode for the output step of the Successive Refinement algorithm

The Successive Refinement algorithm is greedy in the way it trims sub-optimal PLAs from the edges of its consideration. It does not trim sub-optimal regions from the middle, however, and can therefore require more PLAMap runs than is necessary. Typically, several local minima get explored at maximum granularity, providing a good survey of the areas around the minima at a small cost to runtime.

7.2.1.3 Choose N Regions

The Choose N Regions algorithm basically makes a wide sweep of each 1-D space, and then uses the results to iteratively choose N regions to explore at a finer granularity. A region consists of the space between two data points.


```

runPTermSearchSR(circuits, in, pt, out)
{
    leftEdge = 10;    //variables holding the in value of the left
    rightEdge = 90;   //and right edges of our search window
    stepSize = 8;     //current stepSize for taking data
    results[];        //holds my result objects
    bestResult;       //holds the best result at the end

    //While we haven't completed taking data at a step size of 1
    while(stepSize > 0) {

        //For the interval and step size, add the results we don't have
        for(i=leftEdge; i<=rightEdge; i+= stepSize) {
            results = {results, runPLAmap(circuits, in, i, out)};
        }
        //Trim regions of poor results from the left and right edges
        leftEdge = trimFromLeftPT(leftEdge, rightEdge, stepSize, results);
        rightEdge = trimFromRightPT(leftEdge, rightEdge, stepSize, results);
        //halve the step size
        stepSize = floor(stepSize/2);
    }

    //get the best result within the interval
    bestResult = getBestResultInInterval(leftEdge, rightEdge, results);
    //return the PLA variables of the best result
    return {bestResult.in(), bestResult.pt(), bestResult.out()};
}

```

Figure 56. Pseudocode for the product-term step of the Successive Refinement algorithm

Like the Successive Refinement algorithm, the input optimization step of the Choose N Regions algorithm is initiated by taking data points for PLAs with inputs ranging from 4 to 28, but now with a step size of 4. This initially separates the 1-D space into 6 regions, where a region consists of a data point on the left side, a data point on the right side, and the unexplored space between them (see Figure 57). The N best regions are then chosen for further exploration (N=2 was experimentally found to be a good value). The best regions is the region with the best primary result: $\min(\text{leftResult}, \text{rightResult})$. For ties, the region with the best secondary result is taken, as shown in the figure. For the N regions we retain, we halve the step size and acquire data in the center of the chosen

regions: this gives us $2N$ regions (of half the width as before), from which we will again choose the best N regions. The whole process is iterated until N regions have been explored with a step size of 1. The pseudocode for this input step is shown in Figure 58.

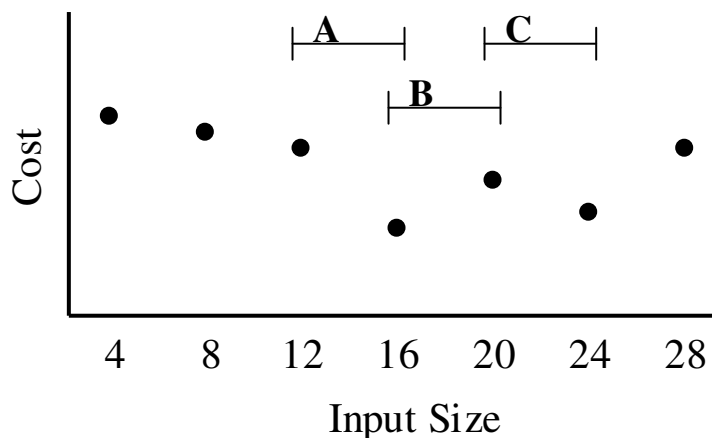


Figure 57. Choose N Regions Algorithm. Region B is the best because it has the best primary point (along with region A) and the best secondary point. Region A is 2nd best, region C is 3rd best

For the output optimization step, we lock the input and product-term values from the best result found in the input step. The output value ranges from 1 to 25, with a step size of 4, and the Choose N process is repeated. For the product-term optimization step, the input and output values from the best result are locked, and the PT values are ranged from 10 to 90 with a step size of 8. After the product-term step has completed its step size of 1, the best overall result is returned. The pseudocode for the output and product-term steps is in Figure 59 and Figure 60 respectively.

The Choose N Regions algorithm has the advantage of retaining, at all steps, N regions of consideration. This allows the algorithm to hone into multiple local minima, as well as throw out old minima that get replaced by new, better results.

```

runInputSearchCN(circuits, N)
{
    results[];           //holds my result objects
    tempResult;         //holds a temporary result object
    region[];           //holds the region objects
    newRegion[];        //holds the newly made region objects

    //Acquire the initial data points
    for(i=4; i<=28; i+=4) {
        results = {results, runPLAmap(circuits, i, 2*i, .5*i)};
    }
    //Build up our list of initial regions
    for(i=0; i<6; i++) {
        region[i] = makeRegion(result[i], result[i+1]);
    }

    //Iteratively find the best N regions, until we hit granularity
    while(1) {
        //Get the best N regions
        newRegion[] = getBestRegions(region, N);
        if(newRegion[0].getSpan() == 1) {
            break; //break when we've hit granularity
        }
        region.empty(); //empty the old list of regions
        //Build the new 2N regions from the N best regions
        for(i=0; i<N; i++) {
            tempResult = runPLAmap(circuits, newRegion[i].middle());
            region[2*i] = makeRegion(newRegion[i].leftResult(), tempResult);
            region[2*i+1] =makeRegion(tempResult, newRegion[i].rightResult());
        }
    }

    //Get the best result, and return the PLA variables
    tempResult = getBestResult(newRegion);
    return {tempResult.in(), tempResult.pt(), tempResult.out()};
}

```

Figure 58. Pseudocode for the input step of the Choose N Regions algorithm

```

runOutputSearchCN(circuits, in, pt, out, N)
{
    results[];           //holds my result objects
    tempResult;         //holds a temporary result object
    region[];           //holds the region objects
    newRegion[];        //holds the newly made region objects

    //Acquire the initial data points
    for(i=1; i<=25; i+=4) {
        results = {results, runPLAmap(circuits, in, pt, i)};
    }
    //Build up our list of initial regions
    for(i=0; i<6; i++) {
        region[i] = makeRegion(result[i], result[i+1]);
    }

    //Iteratively find the best N regions, until we hit granularity
    while(1) {
        //Get the best N regions
        newRegion[] = getBestRegions(region, N);
        if(newRegion[0].getSpan() == 1) {
            break; //break when we've hit granularity
        }
        region.empty(); //empty the old list of regions
        //Build the new 2N regions from the N best regions
        for(i=0; i<N; i++) {
            tempResult = runPLAmap(circuits, newRegion[i].middle());
            region[2*i] = makeRegion(newRegion[i].leftResult(), tempResult);
            region[2*i+1] =makeRegion(tempResult, newRegion[i].rightResult());
        }
    }

    //Get the best result, and return the PLA variables
    tempResult = getBestResult(newRegion);
    return {tempResult.in(), tempResult.pt(), tempResult.out()};
}

```

Figure 59. Pseudocode for the output step of the Choose N Regions algorithm

```

runPTermSearchCN(circuits, in, pt, out, N)
{
    results[];           //holds my result objects
    tempResult;         //holds a temporary result object
    region[];           //holds the region objects
    newRegion[];        //holds the newly made region objects

    //Acquire the initial data points
    for(i=10; i<=90; i+=8) {
        results = {results, runPLAmap(circuits, in, i, out)};
    }
    //Build up our list of initial regions
    for(i=0; i<10; i++) {
        region[i] = makeRegion(result[i], result[i+1]);
    }

    //Iteratively find the best N regions, until we hit granularity
    while(1) {
        //Get the best N regions
        newRegion[] = getBestRegions(region, N);
        if(newRegion[0].getSpan() == 1) {
            break; //break when we've hit granularity
        }
        region.empty(); //empty the old list of regions
        //Build the new 2N regions from the N best regions
        for(i=0; i<N; i++) {
            tempResult = runPLAmap(circuits, newRegion[i].middle());
            region[2*i] = makeRegion(newRegion[i].leftResult(), tempResult);
            region[2*i+1] =makeRegion(tempResult, newRegion[i].rightResult());
        }
    }

    //Get the best result, and return the PLA variables
    tempResult = getBestResult(newRegion);
    return {tempResult.in(), tempResult.pt(), tempResult.out()};
}

```

Figure 60. Pseudocode for the product-term step of the Choose N Regions algorithm

7.2.1.4 Run M Points

The Run M Points algorithm initiates each step by making a wide sweep of the 1-D space, and then iteratively explores points near the best current point. For each 1-D space, the algorithm collects data for M total points before progressing to the next step. Experimentally, a value of $M=15$ was found to provide good results.

Again, the input optimization step starts by taking data points for PLAs with inputs ranging from 4 to 28, with a step size of 4. Next, the best data point is found, and results are taken on either side of it with the largest step size that results in unexplored data points (options are 4, 2, and 1). This is shown in Figure 61. The process is repeated on the best current data point, which is constantly updated, until M data points have been explored for the input step. Once the direct neighbors of a point have been computed, it is eliminated from further explorations; this allows other promising candidates to be explored as well. The pseudocode for the input step is shown in Figure 62.

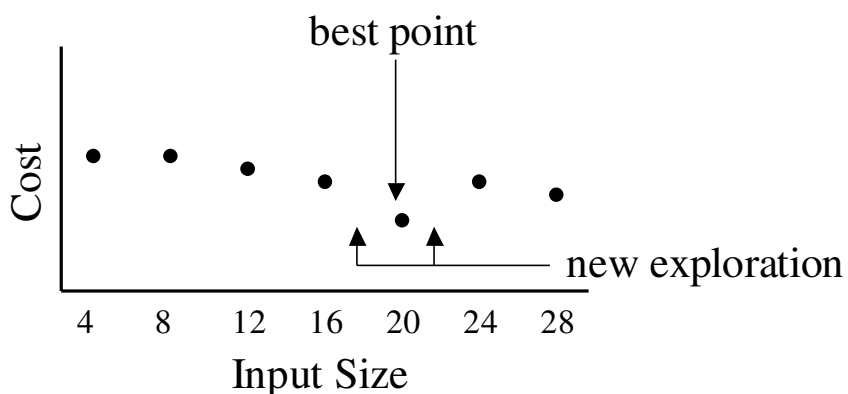


Figure 61. Run M Points Algorithm. The best point is always chosen, and the regions to its left and right are explored

For the output step, we lock the input and product-term values of the best result found in the input step. We then range the output values from 1 to 25, with a step size of 4, and repeat the Run M Points algorithm mentioned above. The product-term step repeats this process, with product-term values ranging from 10 to 90 and a step size of 8 (so possible step sizes are 8, 4, 2, and 1 now). These steps are shown in Figure 63 and Figure 64.

```

runInputSearchRM(circuits, M)
{
    results[];          //holds my result objects
    tempResult;        //holds a temporary result
    pointsTaken = 0;   //how many points we've taken so far

    //Acquire the initial data points
    for(i=4; i<=28; i+=4) {
        results = {results, runPLAmap(circuits, i, 2*i, .5*i)};
        pointsTaken++;
    }

    //Iteratively get the best available point, and run the points
    //next to it. We will only grab results whose search
    //neighborhood has not been exhausted.
    while(pointsTaken <= M) {
        tempResult = getBestResultNotExhausted(results);
        if(tempResult.canRunToLeft()) {
            results = {results, runPLAmap(circuits, tempResult.runToLeft())};
            pointsTaken++;
        }
        if(tempResult.canRunToRight()) {
            results = {results, runPLAmap(circuits, tempResult.runToRight())};
            pointsTaken++;
        }
    }
    //Get the best result, and return the PLA variables
    tempResult = getBestResult(results);
    return {tempResult.in(), tempResult.pt(), tempResult.out()};
}

```

Figure 62. Pseudocode for the input step of the Run M Points algorithm

Because we are exploring to either side of the best result, the range of 10 to 90 is not strictly enforced for the product-term step, as exploration around 10 or 90 would take data points on both sides of the given point. This concept is true for all steps in the Run M Points algorithm. Also note that the input and output steps have the same interval size and step size, while the product-term step has a larger interval and larger step size. The initial sweep of the input and output spaces requires 7 data points, while the initial sweep

of the product-term space requires 11 data points. The product-term sweep is also performed with a step size of 8, whereas the input and output steps use a step size of 4.

```

runOutputSearchRM(circuits, in, pt, out, M)
{
    results[];           //holds my result objects
    tempResult;         //holds a temporary result
    pointsTaken = 0;    //how many points we've taken so far

    //Acquire the initial data points
    for(i=1; i<=25; i+=4) {
        results = {results, runPLAmap(circuits, in, pt, i)};
        pointsTaken++;
    }

    //Iteratively get the best available point, and run the points
    //next to it. We will only grab results whose search
    //neighborhood has not been exhausted.
    while(pointsTaken <= M) {
        tempResult = getBestResultNotExhausted(results);
        if(tempResult.canRunToLeft()) {
            results = {results, runPLAmap(circuits, tempResult.runToLeft())};
            pointsTaken++;
        }
        if(tempResult.canRunToRight()) {
            results = {results, runPLAmap(circuits, tempResult.runToRight())};
            pointsTaken++;
        }
    }
    //Get the best result, and return the PLA variables
    tempResult = getBestResult(results);
    return {tempResult.in(), tempResult.pt(), tempResult.out()};
}

```

Figure 63. Pseudocode for the output step of the Run M Points algorithm

In order to account for the differences in 1-D search space, the product-term step is allowed to explore $(M+6)$ total data points. Of the six additional allowed data points, four of them are to account for the larger initial sweep, and the other two are used to

allow the algorithm to search roughly the same number of regions to the same depth as the input and output steps.

```

runPTermSearchRM(circuits, in, pt, out, M)
{
    results[];          //holds my result objects
    tempResult;        //holds a temporary result
    pointsTaken = 0;   //how many points we've taken so far

    //Acquire the initial data points
    for(i=10; i<=90; i+=8) {
        results = {results, runPLAmap(circuits, in, i, out)};
        pointsTaken++;
    }

    //Iteratively get the best available point, and run the points
    //next to it. We will only grab results whose search
    //neighborhood has not been exhausted.
    //Allow it to run 6 more points than the input and output steps
    //since it is searching a larger 1-D space
    while(pointsTaken <= (M + 6)) {
        tempResult = getBestResultNotExhausted(results);
        if(tempResult.canRunToLeft()) {
            results = {results, runPLAmap(circuits, tempResult.runToLeft())};
            pointsTaken++;
        }
        if(tempResult.canRunToRight()) {
            results = {results, runPLAmap(circuits, tempResult.runToRight())};
            pointsTaken++;
        }
    }
    //Get the best result, and return the PLA variables
    tempResult = getBestResult(results);
    return {tempResult.in(), tempResult.pt(), tempResult.out()};
}

```

Figure 64. Pseudocode for the product-term step of the Run M Points algorithm

While the Choose N Regions algorithm explores N possible optima in parallel, the Run M Points algorithm can be seen as exploring the optima one at a time. It will explore the best optimum until it runs out of granularity, then will turn to the second best

optimum, and so on. In this way it also considers multiple possible optima, as determined by the value chosen for M.

7.2.2 Algorithm Add-Ons

The four algorithms mentioned above comprise the bulk of the Architecture Generator, but some additional routines have been deemed necessary in order to obtain either better or more robust results.

7.2.2.1 Radial Search

As mentioned before, the 3-D search space for this problem is relatively well shaped, but not perfectly so. There are many local optima that might prevent the above algorithms from finding the global optimum. One way to look outside of these local optima is to search the 3-D space within some radius of the current optimum. So for a radius R search around an X-Y-Z architecture, we would vary IN from X-R to X+R, PT from Y-R to Y+R, and OUT from Z-R to Z+R, testing all architectures in this 3-D subspace. Figure 65 shows the pseudocode for this.

We have a strict time constraint on the runtime of the Architecture Generator, so performing the $(2R+1)^3$ extra PLAmapping runs necessary for a radius = R search is not feasible as part of our finalized tool flow. Given looser time constraints and moderately sized circuits, however, small radial searches are not out of the question. Another reason to run radial searches is that it can search a small (but good) part of the 3-D search space exhaustively, and provide an idea of how well the basic algorithms are performing. For this reason, we have performed radial searches of $R = 3$ at the conclusions of the basic algorithms listed above.

```

runRadius(circuit, in, pt, out, R)
{
    result[];    //the result objects we obtain
    tempResult; //for returning the best result

    //Run the radial search
    for(i=(in-R); i<=(in+R); i++) {
        for(j=(pt-R); j<=(pt+R); j++) {
            for(k=(out-R); k<=(out+R); k++) {
                result = {result, runPLAmap(circuits, i, j, k)};
            }
        }
    }
    //get the best result and return it
    tempResult = getBestResult(result);
    return {tempResult.in(), tempResult.pt(), tempResult.out()};
}

```

Figure 65. Pseudocode for the radial search add-on

7.2.2.2 Algorithm Iteration

The Architecture Generator algorithms all assume that the PLAs should be in a 1x-2x-.5x relationship in terms of inputs, product terms, and outputs. This is just a rough guideline, however, and is very rarely the optimal ratio for a given domain. Thus, an interesting idea is to run the basic algorithms and then look at the resulting PLA to obtain a new IN-PT-OUT relationship. A second iteration of the algorithm can be run with this new IN-PT-OUT relationship, exploring the 3-D search space using a relationship that the domain has already been shown to prefer. For example, if the first iteration chose a 10-30-8 architecture, then the IN-PT-OUT relationship for the next iteration would be 1x-3x-.8x. A second iteration has been carried out for all of the algorithms on each domain. Figure 66 shows the general top level pseudocode when a second iteration is used.

```

generalSearchAlgorithmWithIteration()
{
    input circuits;          //circuits in the domain
    output in, pt, out;     //the PLA values for the preferred CPLD arch.
    ptRatio = 2;           //The initial in-pt-out ratio is 1x-2x-.5x
    outRatio = .5;

    for(i=0; i<2; i++) {
        {in, pt, out} = runInputSearchGeneral(circuits, ptRatio, outRatio);
        {in, pt, out} = runOutputSearchGeneral(circuits, in, pt, out);
        {in, pt, out} = runPtermSearchGeneral(circuits, in, pt, out);

        ptRatio = (pt/in);    //get the new ratios for iteration #2
        outRatio = (out/in);
    }
    return {in, pt, out};
}

```

Figure 66. Pseudocode for the top level when using a second algorithm iteration

7.2.2.3 Small PLA Inflexibility

The initial step of each algorithm locks the input value at a value that it deems to be appropriate by testing a wide range of PLA sizes. During the course of algorithms development, we found that domains that migrate to small input values during the input step (i.e. a 4-8-2 PLA) are left with very little flexibility for the corresponding output and product-term steps. The PLAs become strictly input limited, and very few ranges of outputs or product terms will result in reasonable results. When this occurs, the final result of the algorithm tends to be very poor.

To alleviate this, we have added a modification to all of the algorithms. Now, if the input step chooses a PLA with 6 or fewer inputs, the output step will be run both with the PLA found in the input step (6-12-3 or smaller) and with a 10-20-5 PLA. Both of these branches are propagated to the product-term step, and the best overall result of the two branches is taken. We found that this process alleviated the problem of being trapped in small PLA sizes, and it provided better results in all applicable cases. Figure 67 shows the general top level pseudocode that includes this add-on.

```

generalSearchAlgorithmWithSmallPLAInflexibility()
{
    input circuits;      //circuits in the domain
    in, pt, out;        //the PLA values for the preferred CPLD arch.
    in2, pt2, out2;    //temp PLA vars for the alternate path

    {in, pt, out} = runInputSearchGeneral(circuits);
    //run the parallel path if necessary
    if(in <= 6) {
        {in2, pt2, out2} = runOutputSearchGeneral(circuits, 10, 20, 5);
        {in2, pt2, out2} = runPTermSearchGeneral(circuits,in2, pt2, out2);
    }
    {in, pt, out} = runOutputSearchGeneral(circuits, in, pt, out);
    {in, pt, out} = runPTermSearchGeneral(circuits, in, pt, out);
    if(getResult(in, pt, out) < getResult(in2, pt2, out2)) {
        return {in, pt, out};
    } else {
        return {in2, pt2, out2};
    }
}

```

Figure 67. Pseudocode for the top level when using the small PLA inflexibility add-on

7.3 Layout Generator

The Layout Generator is responsible for taking the CPLD architecture description from the Architecture Generator and turning it into a full VLSI layout. It does this by intelligently tiling pre-made, highly optimized layout cells into a full CPLD layout. The Layout Generator runs in Cadence's layoutPlus environment, and uses a SKILL routine that was written by Shawn Phillips [19]. The layouts are designed in the TSMC .18 μ process.

Figure 68 displays a small CPLD that was created using the Layout Generator, along with a floorplan that describes the different sections of the layout. For clarity's sake, the encoding logic required for programming the RAM bits is not shown, but would appear along the left and bottom of the laid out CPLD. Pre-made cells exist for every part of the CPLD: the Layout Generator simply puts together the pre-made pieces as specified by the

architecture description that the Architecture Generator provides. The PLAs are implemented in pseudo-nMOS in order to provide a compact layout.

AND-Plane	Buf	OR-Plane	AND-Plane	Buf	OR-Plane	AND-Plane	Buf	OR-Plane	Buf	OR-Plane
Driver			Driver			Driver				Driver
Xbar2Drive Wire		Reg	Xbar2Drive Wire		Reg	Xbar2Drive Wire		Reg		Xbar2Drive Wire
Crossbar		Reg2Xbar Wire	Crossbar		Reg2Xbar Wire	Crossbar		Reg2Xbar Wire		Reg2Xbar Wire
Xbar2Drive Wire		Reg	Xbar2Drive Wire		Reg	Xbar2Drive Wire		Reg		Xbar2Drive Wire
Driver			Driver			Driver				Driver
AND-Plane	Buf	OR-Plane	AND-Plane	Buf	OR-Plane	AND-Plane	Buf	OR-Plane	Buf	OR-Plane

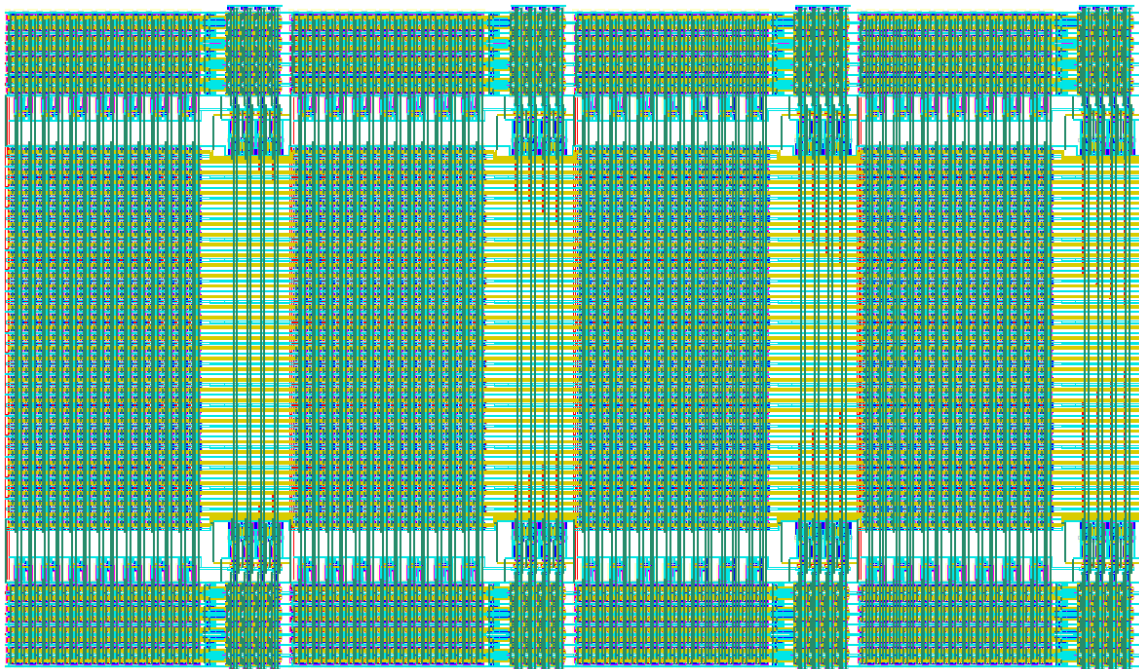


Figure 68. The floorplan of the generated CPLD (top) [25] and the resulting layout (bottom)

7.4 Methodology

We will be creating CPLD architectures that are tailored to specific application domains, but how will we know whether our architectures are actually better than more general architectures? In order to examine this question, we will compare our domain-specific architecture results to some general fixed architectures that we believe to be efficient. All results will be calculated using the same delay and area models that we use for our domain-specific architectures.

We will compare our domain-specific architectures to three different fixed architectures, all of which will use a full crossbar to connect the PLA units in order to conform to our area and delay models. A 1991 analysis of PLA sizing in reprogrammable architectures by Kouloheris and El Gamal [52] showed that PLAs with 8-10 inputs, 12-13 product terms, and 3-4 outputs provide the best area performance for island-style CPLD architectures. While we are not creating island-style CPLD architectures, we would still like to compare our results to the PLA-size proposed in this paper, as it will allow us to model a CPLD architecture that uses relatively small functional units. Therefore, to model this work, the first architecture we will compare to uses 10-12-4 PLAs.

Secondly, our own initial analysis of running several LGSynth93 circuits through PLAmapping showed that CPLDs with roughly 10-20-5 PLAs displayed good performance. We will use this as our second fixed architecture.

Third, we will compare against a XILINX CoolRunner-like architecture. The CoolRunner utilizes 36-48-16 PLAs for its functional units [26]: the choice of a large PLA allows them to provide shallow mappings with predictable and fast timing characteristics. We will therefore compare our domain-specific results to a fixed architecture that uses these PLAs.

Note that we are NOT making a direct comparison to XILINX's CPLDs or any other existing CPLD architecture. By implementing everything using our own physical layouts, we intend to remove the layout designer from the cost equation and simply show

the advantages obtained by using domain-specific architectures rather than implementing designs on fixed architectures.

7.4.1 Failed PLAMap Runs

Our tool flow utilizes an executable of the PLAMap algorithm as part of the architecture generation process. For unknown reasons, PLAMap fails to provide mappings for some circuits on specific architectures. The failures seem to appear at random PLA sizes, although they are sometimes grouped such that a small 3-D region might contain several data points that are unobtainable. The executable we use provides deterministic results, so we have no method of acquiring data for these cases.

Because our algorithms can run for hours or days, we have designed them to raise a visual flag when a failed PLAMap run occurs, but to continue running. In this way, the algorithm continues to accumulate useful data (all PLAMap results are stored on disc to expedite future runs) until we notice that a failed run has occurred, at which point we can manually intervene by stopping the algorithm.

In the case of a failed PLAMap run, we first note the circuit and CPLD architecture of the failure. We then look at the existing PLAMap results for that circuit, and find the result for the architecture that is closest to the size of the failed architecture, requiring that it is no larger than the failed architecture in any PLA variable. The results of this architecture are then substituted for the failed architecture, which allows the algorithm to be restarted. If multiple existing results look like promising candidates, we look at their PLA counts and depths and choose the best result in terms of $PLAs * depth$ (to emulate $area * delay$).

As an example, say that PLAMap failed to provide a result for circuit FOO on an architecture that uses 23-45-8 PLAs. When we notice it, we stop the algorithm. We then look at the existing results for circuit FOO, and find results for 10-20-5, 15-35-20, 20-40-7, and 25-45-8 architectures. The closest matching result that is not larger than the failed architecture is the 20-40-7 result, so we substitute this result for the failed 23-45-8 result. We then restart the algorithm. In the very rare cases where we feel that there is no

reasonable existing result to substitute for the failed result, we run PLAMap offline in order to obtain such a result.

While we haven't collected data on it, we would estimate that PLAMap fails less than 1% of the time. While this might seem insignificant, consider that a domain with 20 circuits that tests 50 architectures will result in 10 failed PLAMap runs if it fails only 1% of the time. In hindsight, it would have been desirable to implement an automatic failure recovery strategy for failed PLAMap runs, as the manual substitution of missing data required significant time.

The process could have been automated by searching for all points within some reasonable radius of the missing data point. The missing data could then be replaced by the best existing data in this radius. Failure to find a reasonable replacement result would cause the algorithm to start running PLAMap on architectures near the missing point, and it would continue until a replacement result is successfully acquired.

7.5 Results

Of the four Architecture Generator algorithms, two of them, the Choose N Regions and Run M Points algorithms, have a user-supplied variable. In the Choose N Regions algorithm we must choose how many regions get explored each iteration, while in the Run M Points algorithm we need to determine how many overall PLAMap runs get executed in each of the three steps.

For the choose N Regions algorithm, the input step and output step both break the 1-D search space into 6 regions, meaning that N can be no larger than 6. We decided to vary N from 1 to 4 when evaluating the algorithm, as this would result in a reasonable number of PLAMap runs. Results for running the Choose N Regions algorithm on the small, medium, and large LGSynth93 domains are shown in Figure 69. The figure shows that gains are achieved by increasing N from 1 to 2, but that further gains are not achieved when setting N to 3 or 4. From this, we determined that $N = 2$ is a good value to use.

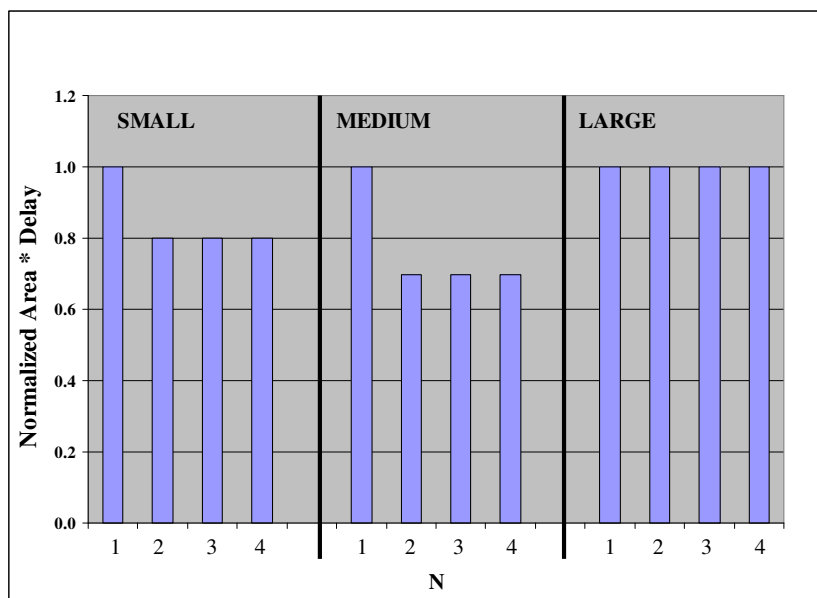


Figure 69. Determination of N in Choose N Regions algorithm

For the Run M Points algorithm, we must determine how many total PLAMap runs are performed for each step of the algorithm. Setting M to 25 would exhaustively search the 1-D input and output spaces, while setting M to 7 would only search the top level without making any interesting descents. We decided that setting M to 10, 15, and 20 would provide a good span of results in a reasonable number of PLAMap runs. The results of running this test on the LGSynth93 domains are shown in Figure 70. The graph shows that M = 15 always outperformed M = 10, and the going up to M = 20 only provided further gains in the small domain, and those gains were very small. From these results, we chose to use M = 15 in all future runs of the Run M Points algorithm.

Now that we have chosen the user-defined variables in the Choose N Regions and Run M Points algorithms, all of the algorithms are fully specified. Next, we took our five main domains and ran each of the four algorithms on each domain. These results are shown in Table 11. All results are normalized to the values obtained for the Choose N Regions algorithm. The columns labeled “Runs” depict how many architectures each algorithm tested for each domain. The bottom row shows the geometric mean for area*delay, and the average for runs.

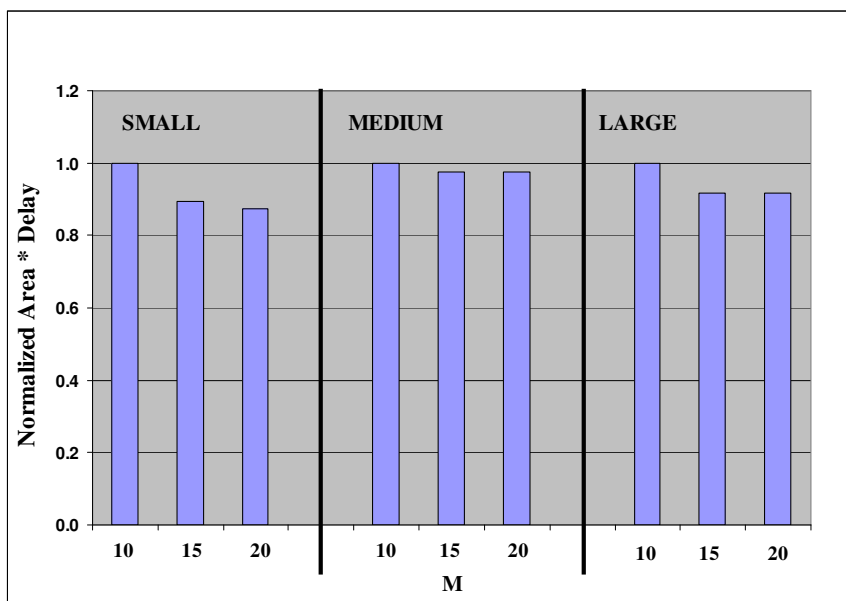


Figure 70. Determination of M in Run M Points algorithm

As Table 11 shows, the Successive Refinement achieved the best results for each of the five domains. The Choose N Regions, and Run M Points algorithms also found the best architecture in most cases, but the simple Hill Descent algorithm only found it for one of the five domains. With respect to runtime, the Hill Descent algorithm took 3.5x to 5.1x fewer runs than the other algorithms, while the Successive Refinement algorithm required the most runs.

Table 11. Architecture results for domain-specific algorithms in terms of area*delay. Results are normalized to the Choose N Regions algorithm, and geometric mean is used for area-delay results

Domain	Algorithms											
	Hill Descent			Succ. Refinement			Choose N Regions			Run M Points		
	Arch	A*D	Runs	Arch	A*D	Runs	Arch	A*D	Runs	Arch	A*D	Runs
Combinational	12-25-4	2.16	13	12-71-4	1.00	90	12-71-4	1.00	40	12-70-4	1.01	52
Sequential	14-27-5	1.18	14	14-38-5	1.00	78	14-38-5	1.00	39	14-38-5	1.00	50
Floating Point	9-26-4	4.50	15	10-24-2	1.00	82	10-24-2	1.00	67	10-24-2	1.00	85
Arithmetic	10-22-2	1.00	14	10-22-2	1.00	76	10-22-2	1.00	67	10-22-2	1.00	85
Encryption	10-20-2	1.47	13	16-67-4	1.00	38	10-25-2	1.47	39	10-25-2	1.47	51
Geo. Mean		1.76	13.8		1.00	69.8		1.08	48.7		1.08	62.6

The Successive Refinement, Choose N Regions, and Run M Points algorithms all chose 4-8-2 PLAs for the floating-point and arithmetic domains in the first step, causing them to be stuck in small PLA architectures. The “Small PLA Inflexibility” algorithm

add-on was applied to these instances to remove them from their sub optimal areas. This caused a slight increase in the number of runs that were needed for these algorithms, most notably in the Choose N Regions and Run M Points results.

Each base algorithm was also run with the second iteration and radial search add-ons described above. Table 12 displays the best architectures found by the base algorithms compared to the best results found using these add-ons. If multiple algorithms found the same result, the algorithm that used the fewest runs is reported.

Table 12. The best base algorithm results compared to best results after a second iteration, and the best results after a radius=3 radial search. Results are normalized area*delay

Domain	Best Base Algorithm				Best with 2nd Iteration				Best with Rad.=3 Search			
	Alg	Arch	A*D	Runs	Alg	Arch	A*D	Runs	Alg	Arch	A*D	Runs
Combinational	CN	12-71-4	1.00	40	CN	12-71-4	1.00	60	CN	9-72-4	0.85	377
Sequential	CN	14-38-5	1.00	39	CN	14-38-4	0.99	77	CN	14-37-6	0.96	377
Floating Point	CN	10-24-2	1.00	67	CN	8-18-2	0.89	105	CN	8-21-2	0.90	307
Arithmetic	HD	10-22-2	1.00	14	CN	10-22-2	1.00	87	HD	7-20-2	0.82	250
Encryption	SR	16-67-4	1.00	38	SR	15-68-4	0.92	75	SR	15-68-4	0.92	375
G Mean / Avg			1.00	39.6			0.96	80.8			0.89	337.2

As Table 12 shows, running a second iteration of the algorithms was able to improve the area-delay product by up to .11x, with a mean area*delay gain of .04x and a mean runtime penalty of 2.0x with respect to the base algorithms. The R=3 radial search add-on was able to reduce the area-delay product by up to .18x, with a .11x mean improvement. The runtime cost for the radius=3 add-on is about 8.5x when compared to the base algorithms. Table 12 shows that running a second iteration can sometimes be as effective as running a radial search, and it requires much less time. Also note that our base algorithms are performing reasonably well, as in all cases they are within .18x of the best results we can easily find.

The radial search add-on found the best architecture for 4 of the 5 domains, but at a significant penalty to runtime. On average, it reduced the area-delay product of the base algorithms by only 11%, but required 8.5x the number of runs. We feel that these performance gains are generally not worth the runtime penalty, and that the radial search add-on should only be used offline as a method of evaluating the results obtained by our faster algorithms.

Running a second iteration was able to improve our performance by 4% at a 2.0x cost to the number of runs required. Note that Table 12 shows that the Hill Descent algorithm was the best base algorithm for the arithmetic domain: if one of the more robust algorithms is reported for this data point, then the cost of running a second iteration is at most 1.6x when compared to the base algorithms. This is a reasonable consideration, as the results in Table 11 display that we would not want to use the Hill Descent algorithm as our base algorithm.

These results show that running a second iteration can provide gains without a drastic penalty to runtime. Closer examination is required, however, to determine which of the algorithms performs best with a second iteration. Table 13 displays the results obtained by running the different search algorithms with a second iteration.

Table 13 shows that, when run with a second iteration, the Successive Refinement algorithm provides slightly better results than the Choose N Regions and Run M Points algorithms. The simpler Hill Descent algorithm is again shown to provide poor results compared to the other algorithms, although it does find them quickly. If only looking at Table 12 and Table 13, we would probably choose the Successive Refinement algorithm with a second iteration as our chosen algorithm. We happen to have data from earlier experimentation, however, which provides slightly different insights.

Table 13. The search algorithms run with a second iteration. Results are normalized area*delay

Domain	Algorithms Run With Second Iteration											
	Hill Descent			Succ. Refinement			Choose N Regions			Run M Points		
	Arch	A*D	Runs	Arch	A*D	Runs	Arch	A*D	Runs	Arch	A*D	Runs
Combinational	12-29-4	1.74	24	12-71-4	1.00	108	12-71-4	1.00	60	12-70-4	1.01	80
Sequential	14-27-5	1.19	21	14-38-4	1.00	151	14-38-4	1.00	77	14-38-4	1.00	99
Floating Point	9-28-5	3.86	25	8-18-2	1.00	116	8-18-2	1.00	105	8-18-2	1.00	135
Arithmetic	10-22-2	1.00	20	10-22-2	1.00	92	10-22-2	1.00	87	10-22-2	1.00	114
Encryption	8-17-2	1.16	25	15-68-4	1.00	75	8-32-2	1.15	78	8-32-2	1.15	101
Geo. Mean		1.56	22.9		1.00	105.5		1.03	80.1		1.03	104.2

Prior to the finalization of our area and delay models, similar data to that in Table 11, Table 12, and Table 13 was compiled using slightly less mature models. This earlier data similarly showed that running a second iteration was a good idea, as it provided 19% performance improvement with respect to the base algorithms. Table 14 shows the

performance of each of the algorithms when run with a second iteration, using these earlier area and delay models.

Table 14. The search algorithms run with a second iteration, results from old area and delay models. Results are normalized area*delay

Domain	Algorithms Run With Second Iteration, Old Models											
	Hill Descent			Succ. Refinement			Choose N Regions			Run M Points		
	Arch	A*D	Runs	Arch	A*D	Runs	Arch	A*D	Runs	Arch	A*D	Runs
Combinational	12-29-4	2.37	24	12-71-4	1.27	109	12-71-4	1.27	60	12-70-4	1.00	99
Sequential	14-27-5	1.19	16	14-38-4	1.19	92	14-38-4	1.19	54	14-38-4	1.00	97
Floating Point	9-28-5	2.97	21	8-18-2	1.00	94	8-18-2	1.00	104	8-18-2	1.00	134
Arithmetic	10-22-2	1.10	18	10-22-2	1.10	69	10-22-2	1.10	76	10-22-2	1.00	133
Encryption	8-17-2	1.43	20	15-68-4	1.00	117	8-32-2	1.00	78	8-32-2	1.00	116
Geo. Mean		1.68	19.6		1.11	94.7		1.11	72.5		1.00	114.7

We again see that the Successive Refinement, Choose N Regions, and Run M Points algorithms all worked well when run with a second iteration. Using this old model, however, it was the Run M Points algorithm that was clearly better than any of the other algorithms. In fact, the architectures shown in Table 14 for the Run M Points algorithm were the best architectures we found for each domain: even running radial searches was unable to improve upon these architectures.

When considering the results in both Table 13 and Table 14, the Run M Points algorithm with a second iteration is shown to be our best algorithm. It provided the best architectures when considered across both the new and old sets of area/delay models, and requires only slightly more runs than the Successive Refinement and Choose N Regions algorithms.

But is it fair to consider results that were obtained from using slightly less mature area and delay models? We think that it is quite reasonable, and in fact that it is a good idea. Our work provides guidelines for how to create domain-specific CPLD architectures, but the methods that we employ are certainly not the only methods available. For example, if our architectures used a CMOS design style rather than a pseudo-nMOS style, it would result in very different area and delay models. We would want our chosen search algorithm to work well on these new models as well as the current models. As such, it makes sense that we would consider multiple sets of area/delay models when choosing the best search algorithm. We have determined that

the Run M Points algorithm with a second iteration is our best algorithm, so all future results will show architectures that were obtained using this algorithm.

7.5.1 Benefits of Domain-Specific Devices

For a given domain, our tool flow provides us with an architecture that efficiently supports the circuits in the domain. This has provided us with architectures for five different domains, but how do these domain-specific architectures compare to some representative fixed architectures? Table 15 compares the performance of our domain-specific architectures with the three representative fixed architectures mentioned above.

Table 15. Performance of domain-specific architectures and fixed architectures. Results are normalized area*delay

Domain	Domain Specific		Xilinx	El Gamal	Observed
			36-48-16	10-12-4	10-20-5
	Arch	A*D	A*D	A*D	A*D
Combinational	12-70-4	1.00	9.40	9.46	4.10
Sequential	14-38-4	1.00	2.28	2.67	2.35
Floating Point	8-18-2	1.00	31.60	14.58	9.00
Arithmetic	10-22-2	1.00	46.49	46.71	18.73
Encryption	8-32-2	1.00	7.43	4.28	3.22
Geo. Mean		1.00	11.85	9.41	5.55

From Table 15 it is apparent that creating domain-specific CPLD architectures is a win over using fixed architectures. For each of the five domains that we considered, the algorithms that we developed always came up with a better CPLD architecture than any of the fixed architectures. Considering the mean performance, the fixed architectures perform 5.6x to 11.9x worse than our domain-specific architectures.

The results shown in Table 15 for implementing the floating point and arithmetic domains on fixed architectures are surprisingly bad, and require some explanation. The mappings of the three largest circuits from the floating-point domain are shown in Table 16. On the Xilinx architecture, the mappings are slightly better than the domain-specific mappings in terms of depth and number of PLAs, but the performance degradation comes from the fact that the Xilinx architecture uses so many PLA outputs, as the size of a CPLD scales quadratically with respect to PLA outputs. The El Gamal and Observed architectures also suffer from the area required by larger PLA output counts, and they

require more PLAs than the domain-specific mapping: CPLDs also scale quadratically with respect to the number of PLAs in an architecture.

Table 16. Mapping results for the three largest circuits in the floating-point domain

	Domain-Specific		Xilinx		El Gamal		Observed	
	8-18-2		36-48-16		10-12-4		10-20-5	
Circuit	PLAs	Depth	PLAs	Depth	PLAs	Depth	PLAs	Depth
FPMult	672	39	464	18	1088	25	762	20
fp_mul	571	50	456	26	1194	32	954	27
fpdiv	423	102	197	53	503	80	440	72

Table 17 displays the mappings of the four largest circuits in the arithmetic domain. The poor results achieved by the Xilinx architecture are again due to the large number of PLA outputs in this architecture, as the Xilinx mappings are similar enough to the domain-specific mappings that they do not overcome this area penalty. The El Gamal and Observed architectures again suffer penalties from both increased PLA output counts, and from an increased number of PLAs in the architecture.

Table 17. Mapping results for the four largest circuits in the arithmetic domain

	Domain-Specific		Xilinx		El Gamal		Observed	
	10-22-2		36-48-16		10-12-4		10-20-5	
Circuit	PLAs	Depth	PLAs	Depth	PLAs	Depth	PLAs	Depth
Div	149	136	174	64	394	84	383	70
MultAddShift	196	29	247	26	854	29	590	29
Mult	276	34	235	19	598	23	540	19
MultBooth3	235	11	93	7	208	10	186	8

Examining these results, it is likely that the fixed architectures in Table 15 are not the best possible fixed architectures for our set of domains. In fact, we have already found the architectures that each domain prefers, so it makes sense that these architectures might work well as fixed architectures. Table 18 shows how these new fixed architectures compare to the architectures obtained using our chosen algorithm. Our domain-specific architecture outperforms the new fixed architectures by 1.8x to 2.5x, even though we have hand selected the fixed architectures to go well with our domains. This shows that even if you manage to pick the best possible domain-generic fixed architecture, there is a bound as to how close you can come to domain-specific results – in this case, domain-specific beats fixed architectures by 1.8x.

Table 18. Results of running each domain on the best domain-specific architectures found. Results are normalized area*delay

Domain	Best	12-70-4	14-38-4	8-18-2	10-22-2	8-32-2
Combinational	1.00	1.00	2.74	3.61	4.38	2.83
Sequential	1.00	2.22	1.00	3.61	2.85	4.13
Floating Point	1.00	3.90	4.24	1.00	1.14	1.02
Arithmetic	1.00	6.21	6.80	1.82	1.00	1.80
Encryption	1.00	1.20	1.28	1.01	1.39	1.00
Geo. Mean	1.00	2.30	2.52	1.89	1.82	1.85

As a final test, we combined the circuits from our five domains into one domain, and ran the architecture generation flow on this new “consensus” domain. While the domain-generic architectures found in Table 18 were found to be effective at supporting specific domains, the architecture that we find for this consensus domain should be effective at supporting all of our domains. Table 19 shows the results of implementing each of our five domains on the architecture found for the consensus domain, which uses 13-79-4 PLAs, and compares these results to the domain-specific results.

Table 19. Results of running each domain on the architecture found for the "consensus" domain. Results are normalized area*delay

Domain	Best Alg	13-79-4
Combinational	1.00	1.09
Sequential	1.00	2.04
Floating Point	1.00	5.18
Arithmetic	1.00	3.49
Encryption	1.00	0.96
Geo. Mean	1.00	2.07

As Table 19 displays, using this new consensus architecture as a fixed architecture is actually less effective than some of the architectures we used in Table 18. The circuits in the encryption domain dominate the size of this new consensus domain, and this has caused the consensus architecture to support the encryption domain very effectively (actually better than our domain-specific architecture), while sacrificing performance on the sequential, floating-point, and arithmetic domains. This data is again showing that, even when the best domain-generic architectures are chosen, they will still perform about 2x worse than domain-specific architectures in terms of area-delay product.

7.5.2 Using Other Evaluation Metrics

We have been running our algorithms with the goal of finding a domain-specific architecture with a minimal area-delay product. While this is a reasonable metric, it is also reasonable to believe that many SoC designers will want to optimize for only area, or only delay. In order to address this issue, we have run our chosen algorithm in both area driven and delay driven modes. Table 20 displays the results of running our chosen algorithm on each of our five domains for area-delay driven, area driven, and delay driven modes. All results are normalized to the area-delay driven results.

Table 20. Results of running our chosen algorithm in area-delay driven, area driven, and delay driven modes for full-crossbar based CPLDs. Results are normalized to area-delay driven values

Full	Area*Delay Driven			Area Driven			Delay Driven		
XBAR	Area	Delay	A*D	Area	Delay	A*D	Area	Delay	A*D
comb	1.00	1.00	1.00	0.97	1.01	0.97	1.12	0.94	1.05
seq	1.00	1.00	1.00	1.06	0.95	1.01	1.18	0.92	1.09
fp	1.00	1.00	1.00	1.00	1.01	1.00	1.55	0.97	1.50
arith	1.00	1.00	1.00	0.99	1.04	1.03	3.52	0.91	3.18
enc	1.00	1.00	1.00	0.99	1.01	1.00	3.41	0.58	1.98
GeoMean	1.00	1.00	1.00	1.00	1.00	1.00	1.90	0.85	1.61

As Table 20 displays, running our algorithm in area driven mode does not have a very large effect on our results, while using delay driven mode gives us noticeably better delay characteristics at a severe cost to area. PLMap primarily attempts to minimize mapping depth (delay), and then applies heuristics that are used to minimize the area of the mapping. Because of this, PLMap always does a relatively good job of minimizing delay, but often fails to minimize the area effectively. This is shown most clearly in the delay driven results, where the delay characteristics of the architecture have been optimized so effectively that the heuristics applied for area improvements were largely ineffective. Both the area-delay driven and area driven results are sensitive to the area of the architecture, and so they both tend toward architectures that are slightly less delay optimized but which are successfully able to apply the area minimization heuristics. This leads the area-delay and area driven results to be similar, as they are both largely driven by the success of the area minimization heuristics.

Table 20 shows that, for full-crossbar based CPLD architectures, using a delay driven metric can provide architectures that use .85x the delay and 1.90x the area of

results obtained using area-delay product as the metric. The table also shows that architectures obtained using an area driven metric are basically equivalent to the architectures obtained using area-delay product.

7.6 Conclusions

This chapter presented a tool flow aimed at tailoring the PLAs in a CPLD to specific application domains. This included an Architecture Generator that could use any of four search algorithms to find a good PLA-size for the architecture, as well as a Layout Generator which tiles pre-made layouts into a full layout.

When compared to realistic fixed CPLD architectures, the domain-specific architectures perform 5.6x to 11.9x better in terms of area-delay product. Even when we hand picked the fixed architectures to be those that were found by our algorithms, our domain-specific architectures still performed at least 1.8x better than fixed architectures. Additionally, iterating the algorithms and performing radial searches around the chosen architectural points show that our algorithms are finding architectures that are within .18x of the best architectures that we can easily find.

We also examined using either area or delay as the evaluation metric, instead of area-delay product. Using an area driven mode provided near equivalent results to using area-delay product, but using a delay driven mode we were able to reduce the delay by 15% when compared to area-delay driven results. This is because we were able to find architectures that PLAMap successfully optimized for depth, at a slight cost to area.

Running the Run M Points algorithm with a second iteration provided the best results in terms of performance, at only a small increase to runtime over other algorithms. The Run M Points algorithm has thus been chosen as the preferred method for tailoring CPLD logic to an application domain, and will be used exclusively for future explorations.

8 Routing in Domain-Specific CPLDs

In the previous chapter we presented a tool flow that creates crossbar-based CPLDs that are tailored to specific application domains. Those CPLDs are interconnected using a crossbar interconnect that is fully populated with switches. Full crossbars require a large amount of area, since their size scales quadratically with respect to PLA count. This area inflation is one reason that commercial CPLDs don't come in very large sizes.

The area effects of using crossbar-based CPLDs can be seen in Table 21. Table 21 lists the areas of the domain-specific architectures that we found in chapter 7 normalized to the sequential domain, along with the percentage of area devoted to the CPLD's routing resources. As the table shows, the routing area is reasonable for quite small architectures (only 61.5% for the sequential domain), but as architectures get larger the percentage of the area that is used for routing becomes prohibitively large. For the encryption domain, the routing resources utilize 75x more area than the logic resources.

Table 21. Area required by routing resources for crossbar-based CPLDs from the previous chapter

Domain	Normalized Area	Routing Area
Sequential	1.00	61.5%
Arithmetic	21.03	93.0%
Combinational	22.12	84.5%
Floating Point	97.26	96.8%
Encryption	1006.85	98.7%

This chapter presents a method for reducing the size of the CPLD's routing structure by reducing the number of switches in the crossbars. For the remainder of this section, we will define a crossbar as the connection matrix that provides connectivity between the general interconnect wires and the input wires of a PLA. Figure 71 displays this definition, showing a full crossbar on the left and a sparse crossbar on the right. Note that this crossbar is repeated for each PLA in the CPLD architecture. In this section we will be creating sparse crossbars for our CPLDs, which will provide area and delay gains

over full crossbars, but will also require the development of techniques to ensure routability.

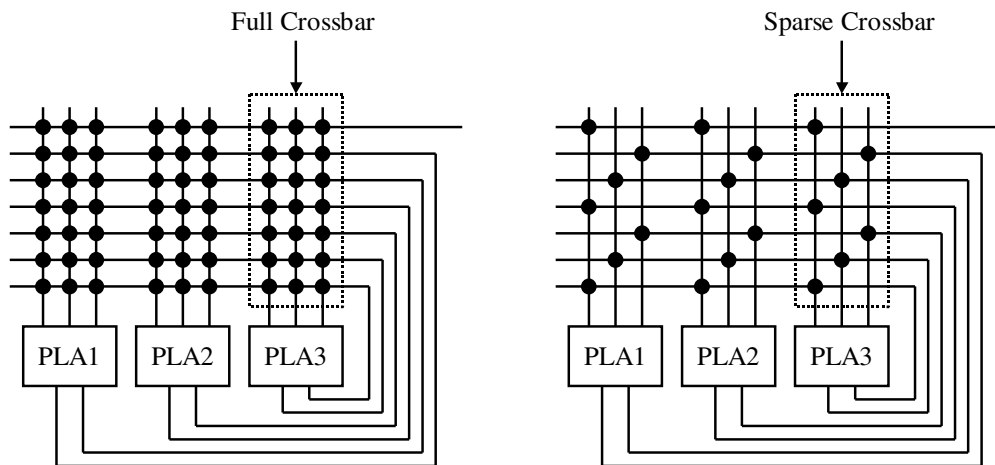


Figure 71. A full crossbar (left) and a sparse crossbar (right). Note that the crossbar gets repeated for each PLA in the CPLD architecture

8.1 Crossbars

The advantage of using a full crossbar is that it provides full connectivity between input and output wires, which makes the routing of signals through the crossbar trivial. As Figure 72 shows, each output wire in a full crossbar can be connected to any input wire, and multiple outputs can connect to the same input.

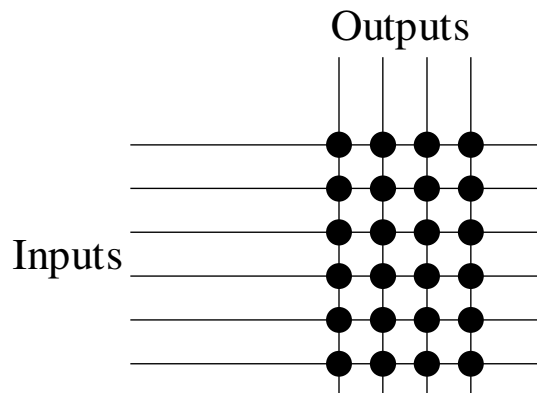


Figure 72. In a full crossbar, output lines can be connected to any input line

A simple full crossbar with n inputs and m outputs requires $n*m$ switches. The capacity of a crossbar, c , is the largest number of inputs in a crossbar that are always routable to outputs. A full crossbar provides the maximum capacity possible, as $c=m$.

In the CPLD architectures that we create, the outputs of the crossbars are directly connected to PLA inputs. One characteristic of PLAs is that their inputs are permutable, so it doesn't matter which input any given signal arrives on, just that it arrives. This means that full crossbars provide more flexibility than we need, as our only requirement is that the crossbars provide full capacity.

Crossbars can provide full capacity without requiring switches in every location. This can be done with a "minimal full-capacity" (abbreviated *minimal* from here forward) crossbar, like that shown in Figure 73. In the figure, the top 4 input wires can only be switched onto a specific output wire, while the remaining input wires are fully connected to the outputs. This style can be generalized for a crossbar with n inputs and m outputs by saying that each of the top m inputs can be switched onto a specific output wire, while the bottom $(n-m)$ inputs will be fully connected to the outputs.

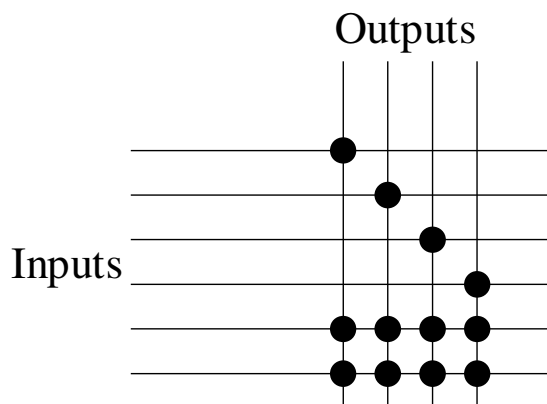


Figure 73. A minimal crossbar that provides full capacity

The number of switches required in a minimal crossbar is shown in Equation 19 (REF). As can be seen from the equation, if $m \approx n$ then p will be relatively small. For crossbars that are nearly square, therefore, it is efficient to provide full connectivity using one of these minimal crossbars. In our crossbars, however, we have a situation where $n \gg m$. Plugging this into Equation 19, it is apparent that the number of switches in a

minimal crossbar approaches $n*m$, and there is very little advantage gained from using the minimal crossbar.

$$p = (n - m + 1) * m \quad (19)$$

It is clear that we will not be able to decrease the sizes of our crossbars while maintaining full capacity. Crossbars that do not provide full capacity are called “sparse” crossbars.

8.2 Sparse-Crossbar Generation

When creating sparse crossbars, the objective is to maximize the routability of the crossbar for a given switch count, where routability is defined as the likelihood that an arbitrarily chosen subset of inputs can be connected to outputs. While switch placement is deterministic for crossbars that provide full capacity, it is not obvious how the switches should be placed in a sparse crossbar in order to maximize routability. This problem, though, has been effectively addressed by a sparse-crossbar-generation tool created by Lemieux and Lewis [53].

The switch placement problem can be generalized by saying that subsets of input wires should span as many output wires as possible in order to provide maximum routability. So for a given set of input wires, we want to maximize the number of output wires that can be reached by those input wires. This makes intuitive sense, as providing a larger set of reachable outputs should make it more likely that each input can be assigned to a unique output. Figure 74 gives an example of this. In the left crossbar, every subset of four input wires spans all four output wires, and it turns out that this crossbar provides a capacity of 4. In the crossbar on the right, the subset {a, b, e, f} spans only three output wires, and therefore cannot be routed. This crossbar only has a capacity of 3.

Equation 20 displays how many input wire subsets exist for a crossbar with n inputs. Clearly, we cannot maximize the span of this many subsets if n is even marginally large. Because of this, we will simplify the problem to only considering input wire pairs. This is a reasonable simplification to make, because we can find the span of large input groups simply by unioning the spans of smaller input groups. An example of this is shown in

Equation 21. By spreading out the switches between every pair of input wires we will maximize our routability, while only having to consider $O(n^2)$ subsets.

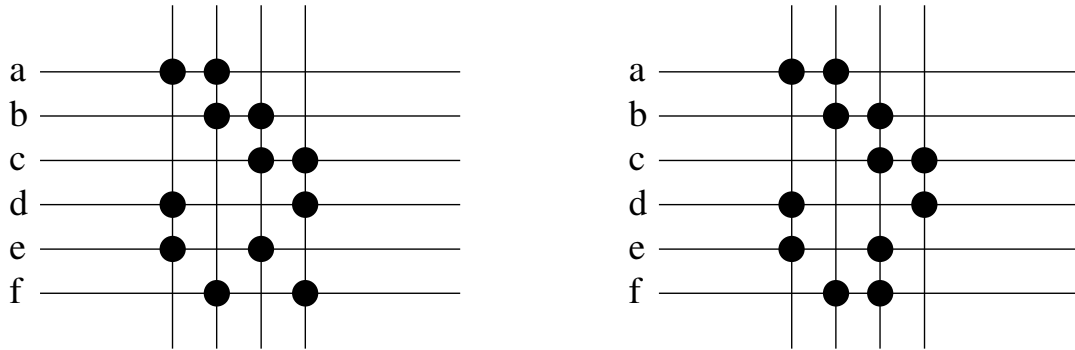


Figure 74. The crossbar on the left provides $c=4$, while the crossbar on the right only provides $c=3$. Only one switch is in a different location on the right (on wire f), but it reduces the number of outputs that are reachable by the input subset $\{a, b, e, f\}$

$$\sum_{l=2}^n {}^n C_l \tag{20}$$

$$outputSpan\{a,b,c,d\} = outputSpan\{a,b\} \cup outputSpan\{c,d\} \tag{21}$$

Spreading out the switches between every pair of input wires is very similar to the strategy used for creating communication codes. In fact, our problem can be mapped into the communication code problem very easily. The locations where switches are placed on an input wire can be represented by a vector of 1s (where switches are present) and 0s (not present). Each vector is of length m , and there are n such vectors. An example of this is shown in Figure 75.

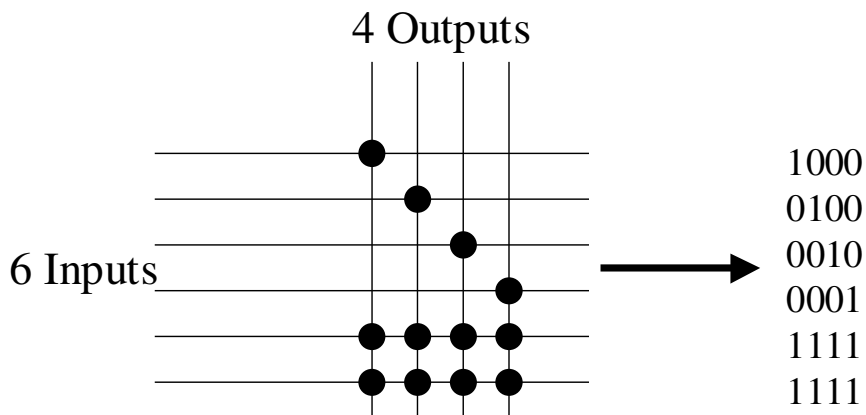


Figure 75. Representing the input lines as vectors of 1s and 0s

When represented in this fashion, spreading out the switches between two input wires is the same as maximizing the Hamming distance between the wires. The Hamming distance between two vectors is the sum of the bitwise XOR of the vectors. Figure 76 gives some examples of switch placements, their vector representations, and the resulting hamming distances. It is clear from the figure that the hamming distance increases as the switches are spread out among the output wires.

Crossbar	Vectors	Hamming Distance
	1001 0110	4
	1010 0110	2
	0110 0110	0
	1010 0111	3

Figure 76. Some crossbars, their vector representations, and their hamming distances

Maximizing the minimum Hamming distance between vectors is a common code design technique, and the cost function in Equation 22 is one that has been used in practice [54]. In the equation, bv_x and bv_y represent the bit vectors for input rows x and y respectively. An efficient switch placement can be found by minimizing this cost function across all the vectors representing the crossbar input lines.

$$SwitchPlacementCost = \sum_{\forall x, y | x \neq y} \frac{1}{HammingDistance(bv_x, bv_y)^2} \quad (22)$$

One thing to note about Equation 22 is that, if the hamming distance of two vectors is 0, then the cost is undefined. This works fine for communication codes, as it is illegal to have two vectors with a hamming distance of 0 because the codes would be

indistinguishable from each other. In our application, however, two input lines can have the same switch patterns and still be useful, so we must define a cost for this situation. Moving from a hamming distance of 2 to 1 causes the cost to change by a factor of 4 (from $\frac{1}{4}$ to 1), so we decided that it would be reasonable for a move from 1 to 0 to change the cost function by the same factor. Vectors with a hamming distance of 0 therefore incur a cost of 4.

This rest of this subsection will introduce our implementation of the sparse-crossbar-generation algorithm from [53]. The goal of the algorithm is to create a sparse crossbar of maximum routability given the values n (inputs), m (outputs), and p (switches). The top level pseudocode for this process is shown in Figure 77.

```

switchPlacementAlgorithm()
{
    input n, m, p;      //crossbar has n inputs, m outputs, p switches
    output sw[n][m];   //switch placement matrix

    initialSwitchPlacement(n, m, p, sw);
    makeSwitchMoves(n, m, sw);
    smoothSwitches(n, m, p, sw)

    return sw
}

```

Figure 77. Top level pseudocode for the switch placement algorithm

8.2.1 Initial Switch Placement

To initiate the algorithm, p switches must be placed such that they are evenly distributed both on the input and output lines. We define the number of switches connected to an input wire is its *fanout*, and the number of switches connected to an output wire is its *fanin*. To be evenly distributed, we require that the *fanout* of any input line differ from any other input line by no more than 1. Similarly, we require that the *fanin* of any output line differ from any other output line by no more than 1. Requiring a

smooth switch distribution will help us obtain good routability, as each line will be roughly as connected as any other line. The initial switch placement is performed deterministically, using the algorithm in Figure 78.

```

initialSwitchPlacement(n, m, p, sw) // n inputs, m outputs, p switches
{
    fanin = ceil(p/m);
    switches_left = p;
    switches_placed_on_output = 0;
    current_in = 0;
    current_out = 0;

    //initialize switch matrix to 0
    for(i=0; i<n; i++) {
        for( j=0; j<m; j++) {
            sw[i][j] = 0;
        }
    }

    //place switches in matrix such that they are balanced
    //across inputs and outputs respectively
    while(switches_left > 0) {
        sw[current_in][current_out] = 1;
        switches_left--;

        //cycle through inputs, top to bottom
        if(++current_in == n) {
            current_in = 0;
        }

        //cycle through outputs, left to right, obeying fanin
        if(++switches_placed_on_output == fanin) {
            current_out++;
            switches_placed_on_output = 0;
            fanin = ceil(p/m);
        }
    }
}

```

Figure 78. Pseudocode for initial switch placement algorithm

An example of an initial switch placement is shown in Figure 79, which displays a crossbar with 12 inputs, 8 outputs, and 53 switches. One thing to note about the initial placement is that output lines on the left will receive more switches than those on the right if $p\%m \neq 0$. This will not have any negative impact on our routing, however, since the crossbar outputs directly feed the PLA inputs, which are freely permutable. For example, nothing is lost by making *PLA0_Input1* slightly easier to route to than *PLA0_Input15*.

The initial placement algorithm also works such that input lines near the top will receive more switches than those on the bottom if $p\%n \neq 0$. If the crossbar remained like this, the PLAs that feed their outputs to the upper interconnect wires (and thus the upper crossbar wires) would have a routing advantage over PLAs that feed their outputs to lower interconnect wires. This potential problem is alleviated by the switch smoothing algorithm described in section 1.2.4, which redistributes the crossbar such that crossbar input lines with higher fanout are not grouped together.

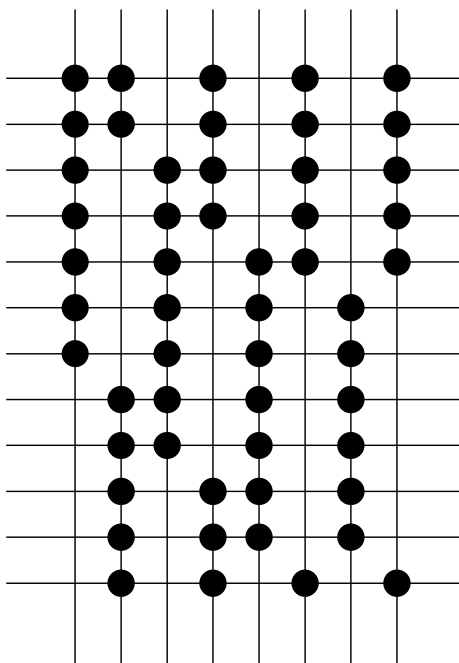


Figure 79. The initial switch placement for a 12-input, 8-output, 53-switch crossbar

There is a negative impact from this uneven switch fanout, however. Different horizontal routing tracks will see different numbers of switches in their signal paths, and will thus have slightly different delay characteristics. Since we repeat the same sparse crossbar at each PLA within the CPLD architecture, this difference in switch fanout is multiplied by the number of PLAs in the architecture. Thus, across the span of the CPLD, some horizontal routing tracks can have *PLA_Count* more switches on them than other tracks.

8.2.2 *Moving Switches*

The switches are initially placed in a very regular manner, so in order to obtain good routability from our sparse crossbar we must be able to move the switches around. We must be careful, however, to move the switches in such a way that they are still evenly distributed among the input and output lines. In order to obtain this behavior, we ensure that switch movements do not change the number of switches that exist on an input or output line.

Figure 80 displays the two switch movements that we allow. At the intersection of two input lines and two output lines, if switches exist on one of the diagonals but not on the other diagonal then the switches can be moved as shown. This ensures that each of the input and output lines retains the same number of switches.

Figure 81 shows the switch movement algorithm as implemented. When a switch move is proposed, the cost function is updated by recalculating the hamming costs involving only the vectors being altered. Lemieux and Lewis initially used simulated annealing to conditionally accept moves based on cost and temperature criteria, but they found that negative cost moves were nearly always found again and undone, and that a greedy scheme worked just as well. We therefore reproduce this, and use a purely greedy strategy for accepting moves.

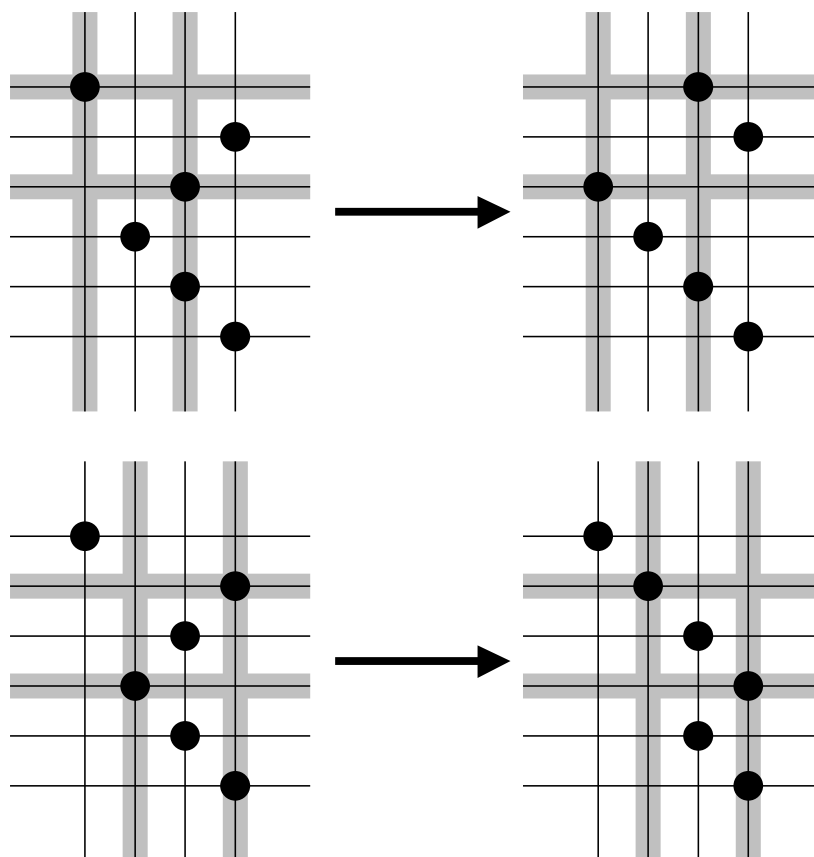


Figure 80. Allowable switch moves. Switches must occur on one diagonal of the intersecting lines, but not on the other diagonal

Figure 82 provides evidence that the algorithm developed by Lemieux and Lewis is in fact providing highly routable sparse crossbars. The graph shows the routability of various 80-input, 12-output, 160-switch sparse crossbars. The x-axis shows the test vector size (number of inputs), and the y-axis shows the percentage of vectors that were successfully routed at each size.

```

makeSwitchMoves(n, m, sw)
{

    possible_outs[]; //output lines with legal moves - initially empty

    for(i=0; i<(n*m); i++) { //try n*m moves

        n1 = chooseRandomInputLine(sw); //get first input line

        while(possible_outs.isEmpty()) {
            n2 = chooseRandomInputLines(sw); //get second input line
            possible_outs = identifyPossibleOutputMoves(n1, n2); //legal moves
        }

        //get output lines
        m1 = chooseRandomOutputLine(n1, n2, sw, possible_outs);
        m2 = chooseRandomOutputLine(n1, n2, sw, possible_outs);

        //while they don't make a legal move, change m2
        while(!isLegalMove(n1, n2, m1, m2)) {
            m2 = chooseRandomOutputLine(n1, n2, sw, possible_outs);
        }

        preCost = cost(n1, n2, sw); //initial cost

        //move the chosen switches
        moveSwitches(n1, n2, m1, m2);

        postCost = cost(n1, n2, sw); //new cost

        //undo the move if it was bad
        if(postCost > preCost) {
            moveSwitches(n1, n2, m1, m2);
        }

        possible_outs.makeEmpty(); //reset our possible moves to empty

    }
}

```

Figure 81. Pseudocode for the switch movement algorithm

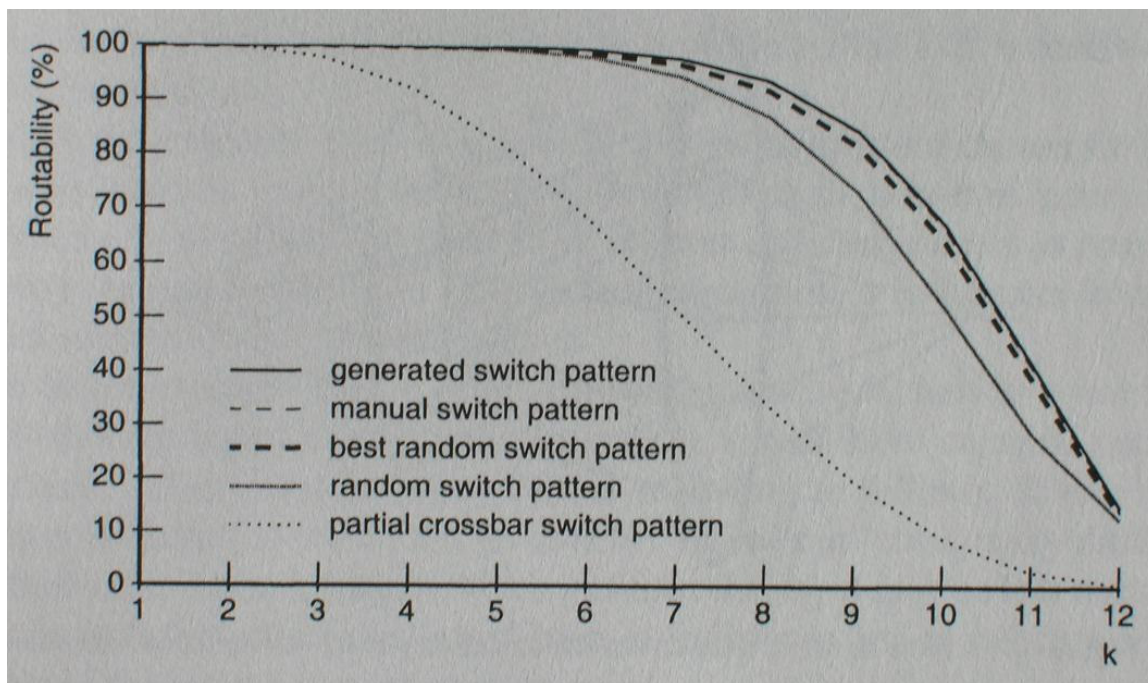


Figure 82. Routability results from several different switch patterns [53]

The most successful crossbar was created by the algorithm described herein, beating out a manual switch pattern sparse crossbar that was meticulously hand-generated by a PLD designer. Also included on the graph are lines for the best random switch pattern generated (out of a large set), the average across several random switch patterns, and a partial-crossbar pattern akin to that used in the HP Plasma architecture. These results give us confidence that we are getting efficient crossbars out of our sparse crossbar generation tool.

8.2.3 Algorithm Termination

The algorithm in [53] allows the user to specify how many switch movements are attempted. Our process is automated, however, so we needed to determine how many switch moves should be attempted as a function of n , m , and p in order to obtain a good sparse crossbar. Experiments showed that attempting $n*m$ switch moves provided good results with a reasonable runtime: beyond this number of moves, the tool was never able

to improve the cost function by more than an additional 1% regardless of how many moves were attempted.

8.2.4 Switch Smoothing

The resulting sparse crossbar has a switch distribution that is going to look fairly random. Because of this, there are likely to be regions in the crossbar that have relatively high switch densities, as well as regions that have relatively low switch densities. The preceding algorithm has ensured that our crossbar is highly routable, but it has not ensured that the switches are spread out in such a manner that they will lead to an efficient layout. This issue of switch smoothing was not considered in the work done by Lemieux and Lewis [53].

When considering the layout of a crossbar, it is clear that the switches and their corresponding SRAM bits determine the area required by the crossbar, as the input and output wires can simply be brought in on metal layers above these devices. In order to achieve a compact layout, therefore, it is desirable to pack the switches as closely as possible.

When creating full crossbars, it is easy to route the input lines to the switches because there is a switch at the intersection of every input and output wire. Figure 83 shows this in cartoon fashion. When we remove switches from the crossbar, however, there are additional considerations. First of all, the input wires will need to be closer together since the overall height of the crossbar will decrease (due to fewer switches). Second, the input wires may now need to connect to switches that are not directly below the wire: vertical jogs will be required to connect wires to switches that are either north or south of the input wire's horizontal track. Figure 84 displays this detail.

In order to keep the switches packed tightly, we must limit the number of vertical jogs that are required in our sparse crossbars. Otherwise, the pitch required by the vertical jogs would start to dominate the east-west dimension of the crossbar layout. If the layout contains N input wire pitches per vertical switch pitch, then we can minimize the number of vertical jogs by requiring that each output line be attached to exactly one

switch for every N input lines. This will ensure that each input line attaches to a switch that is underneath it. An example of this is shown in Figure 85.

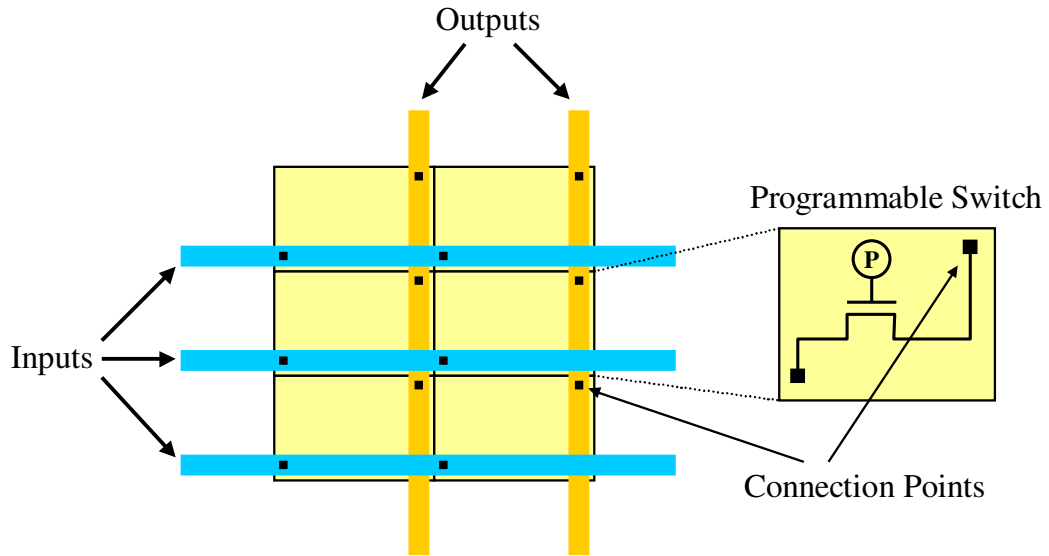


Figure 83. Routing full crossbars is easy because there is a programmable switch at the intersection of every input and output line

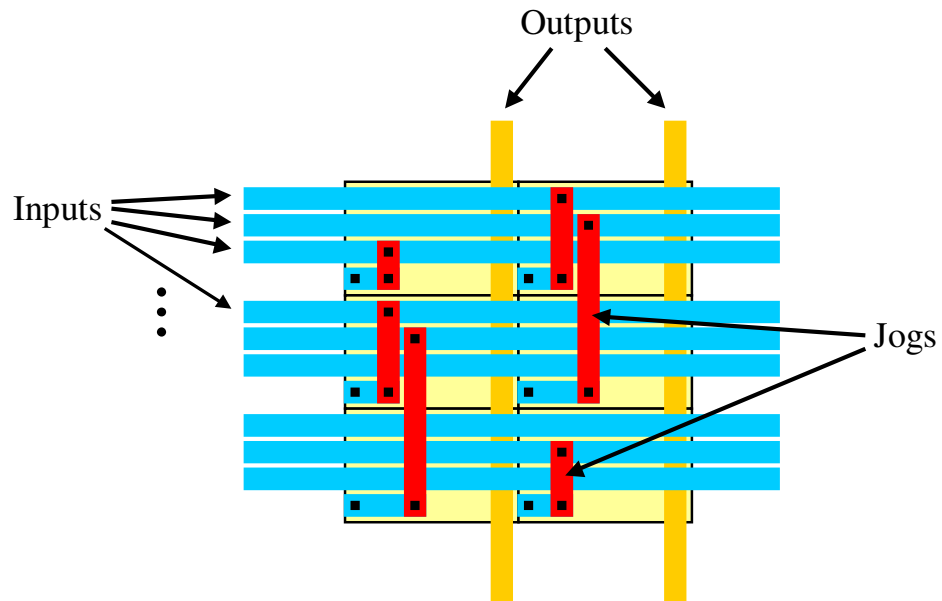


Figure 84. Routing sparse crossbars requires us to pack the input lines closer together and to add wire jogs in order to connect input lines to the proper switches

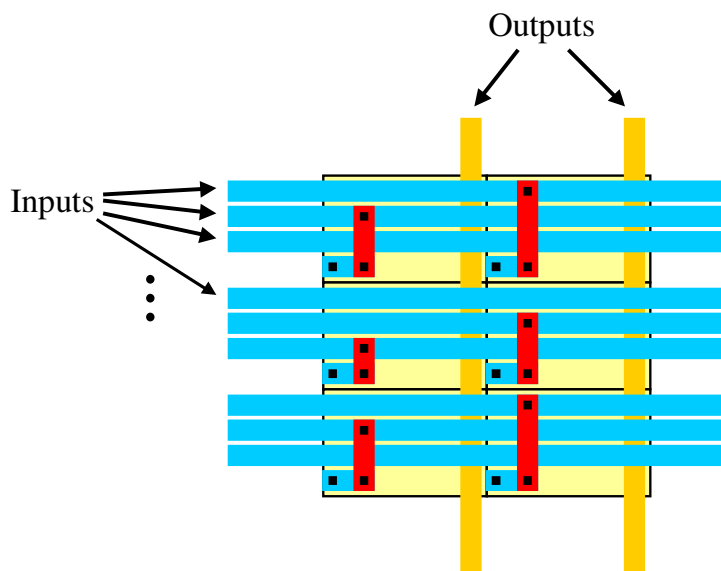


Figure 85. If each output line is attached to exactly one switch per every N input lines, then every input line will connect to a switch directly below it, and only one pitch of jogs is required

Since we are creating an effectively random switch placement in our sparse crossbars, it is unlikely that they will align such that each output has one switch per every N input lines. The input lines in our sparse crossbar can be permuted, however, so we can move them around in order to enforce this property. The pseudocode for accomplishing this is shown in Figure 86.

The basic principle of the algorithm in Figure 86 is to order the input lines from north to south such that, after placing input line P , there have been P/N switches placed on each output line (for N input lines per switch pitch). This is achieved by minimizing the cost function shown in Equation 23 as each input line is placed. In this equation for an n input, m output sparse crossbar, x is the current output line, S_x is the number of switches that have been placed in output line x , P is the current input line, and N is the number of input lines per switch pitch in the layout. Figure 87 shows an example of running the switch smoothing algorithm on a sample crossbar, resulting in a smoothed crossbar. Figure 88 shows the layouts derived from the unsmoothed and smoothed crossbars from Figure 87, displaying the improvement in the lengths and pitches of the required wire jogs.

```

smoothSwitches(n, m, p, sw)
{
    //matrix to hold the smoothed switch matrix
    ordered_sw[n][m];

    //how many switches have been placed on each output line
    swCount[m];
    for(i=0;i<m;i++) { swCount[i]=0; }

    //whether we've place an input line yet
    inPlaced[n];
    for(i=0;i<n;i++) { inPlaced[i]=0; }

    //switch density
    swDens = p/(n*m);

    tempCost;

    for(i=0;i<n;i++) { //must place n smoothed input lines
        tempInput = -1;
        bestCost = infinity;
        for(j=0;j<n;j++) { //check which of n lines is next
            //Calculate the cost of inserting input line j next
            tempCost = smoothCost(sw[j], I, swCount, swDens);
            if(tempCost < bestCost) {
                if(!inPlaced[j])
                    tempInput = j;
                bestCost = tempCost;
            }
        }
        ordered_sw[i] = sw[tempInput]; //set proper line
        for(k=0;k<m;k++) {
            swCount[k] += ordered_sw[i][k];
        }
        inPlaced[tempInput] = 1;
    }

    //set the switch matrix to the smoothed result
    sw = ordered_sw;
}

```

Figure 86. Pseudocode for the switch smoothing algorithm

$$\sum_{\forall x} (S_x - \frac{P}{N})^2 \tag{23}$$

Unsmoothed Crossbar	Input Line (<i>P</i>)	Smoothed Crossbar	<i>S_x</i> Matrix	Cost	Progress
0 0 0 1	1	0 0 0 1	0 0 0 1	.84	↓
0 0 1 1	2	0 1 1 0	0 1 1 1	.76	
0 1 0 1	3	1 0 0 0	1 1 1 1	.64	
1 0 0 1	4	0 0 1 1	1 1 2 2	1.04	
0 0 1 0	5	1 1 0 0	2 2 2 2	.00	
0 1 1 0	6	0 0 1 0	2 2 3 2	.84	
1 0 1 0	7	0 1 0 1	2 3 3 3	.76	
0 1 0 0	8	1 0 0 1	3 3 3 4	.76	
1 1 0 0	9	1 0 1 0	4 3 4 4	.84	
1 0 0 0	10	0 1 0 0	4 4 4 4	.00	

Figure 87. Each step in the switch smoothing algorithm is displayed for the smoothing of the crossbar on the left

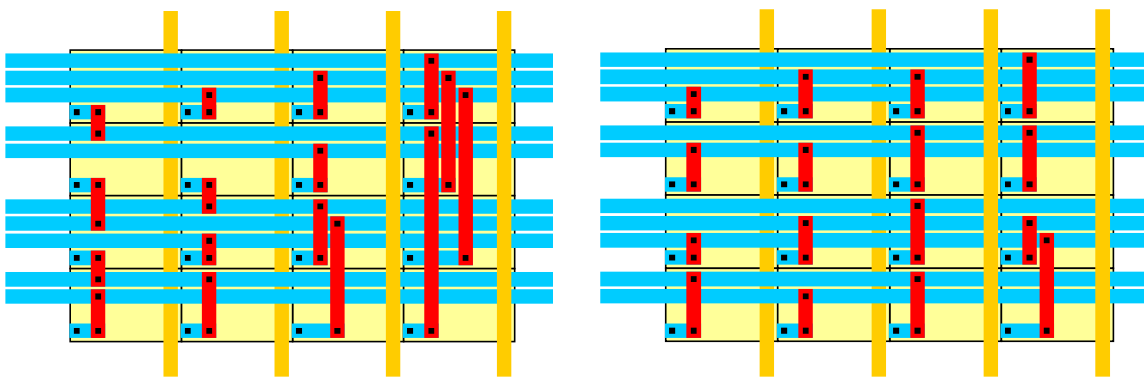


Figure 88. Representative layouts of the unsmoothed (left) and smoothed (right) crossbars from Figure 87. Notice that the switch smoothing algorithm has reduced the jog pitch to near minimum, and has reduced the jog congestion significantly

We have chosen a layout style that restricts us to 8 or fewer jog pitches per crossbar output. This means that any crossbar requiring more than 8 jog pitches would force us to spread out our switches in the east-west direction, wasting area in our final layout. Table

22 displays the pre-smoothing and post-smoothing jog pitches required of the sparse crossbars that we created for our five domains (the performance of these architectures appears later in this section). As the table shows, four of the five crossbars would have resulted in area penalties in the final layouts if the crossbar-smoothing algorithm had not been applied. Also note that the final smoothed crossbars easily meet our requirement of 8 or fewer jog pitches.

Table 22. Results of running the switch smoothing algorithm on crossbars acquired for each domain

Domain	Crossbar			Required Jog Pitches	
	Inputs	Outputs	Switches	Pre-Smoothing	Post-Smoothing
Sequential	155	14	293	6	2
Arithmetic	613	21	1216	12	3
Combinational	735	14	2647	19	2
Floating Point	1574	18	4627	17	3
Encryption	5002	23	32099	48	3

8.3 Routing for CPLDs with Sparse Crossbars

Our sparse crossbars do not provide full capacity, so we will have to use a routing algorithm in order to ensure that signals can reach their destinations. PLAmapping provides the necessary mapping information, which includes the inputs and outputs of each PLA in the CPLD mapping. The inputs and outputs of a PLA are completely permutable, and we are using a complete network implementation, so this results in a very simple routing graph. Only two types of routing decisions need to be made: which horizontal routing track a PLA output will connect to, and which PLA input track a horizontal routing track will connect to. Figure 89 shows a simple CPLD with sparse crossbars, and the corresponding routing graph for the architecture.

We decided to use the Pathfinder [55] routing algorithm for our CPLD architectures. All of our resources are of similar delay, so we can use their negotiated congestion router, which does not account for delay. The corresponding cost function for using a resource is shown in Equation 24, where b_n is the base cost, h_n is the history cost, and p_n is the sharing cost. We set b_n to 0 so that the cost function simply considers $h_n * p_n$, and we allow the algorithm to run for 1000 iterations before judging a mapping to be unroutable

with a given sparse crossbar. We can use an extremely large number of iterations because of how simple our routing graphs are: a single iteration of the pathfinder algorithm on our largest architecture takes only a couple seconds.

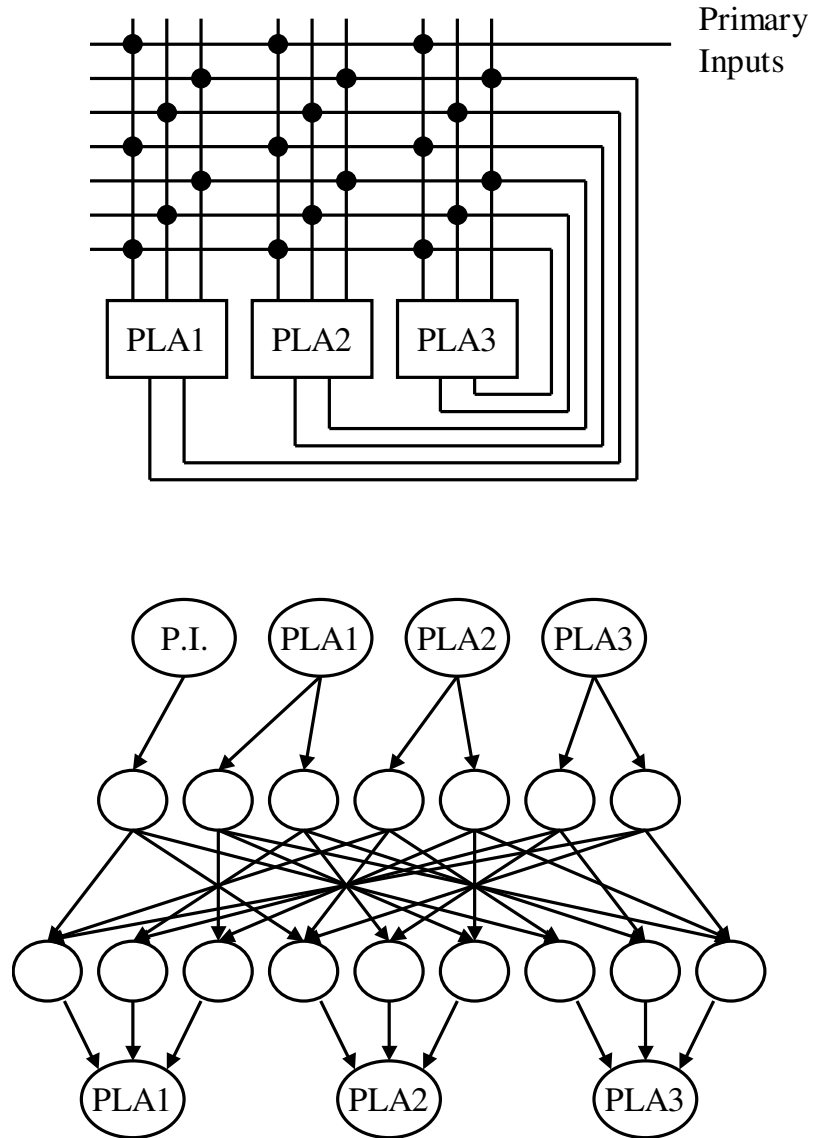


Figure 89. A simple CPLD and its routing graph

$$cost_n = (b_n + h_n) * p_n \tag{24}$$

8.4 Determining Switch Density of Sparse Crossbars

In order to determine how many switches our crossbars need, we perform a binary search on the crossbar switch count. This is shown in pseudocode in Figure 90. For an n input, m output crossbar, the minimum switch count is n and the maximum switch count is $n*m$. Starting with a crossbar with $.5(n*m)$ switches, we iteratively create a sparse crossbar and route on the specified CPLD architecture, adjusting the switch count according to whether our route fails or succeeds. We do this to maximum granularity, which requires only $\log(n*m)$ iterations of the router.

8.5 Results

Our goal in using sparse crossbars is to reduce the overall area that our CPLD architectures require, so we should start by examining how much of a reduction we can obtain. Table 23 displays the area, delay, and area-delay product results obtained by using sparse crossbars in the CPLD architectures that were found in the previous chapter. The table shows that CPLDs with sparse crossbars require only 40% of the area that CPLDs with full crossbars require, validating our use of sparse crossbars. Removing connectivity from the crossbars also results in improved delay performance due to decreased capacitance on our interconnect paths. The table shows that the delay of sparse-crossbar-based CPLDs is only 40% that of full-crossbar-based CPLDs. Combining the two terms to form area-delay product, we see that sparse-crossbar-based CPLDs require only .16x the area-delay product of full-crossbar-based CPLDs.

One surprising result from Table 23 is that the area reductions obtained by using sparse crossbars seem to be independent of the routing area of the full-crossbar CPLD architecture. Intuitively, it seems that an architecture whose routing fabric requires more of the CPLD area would have more to gain from using sparse crossbars, suggesting that the encryption domain (98.7% routing) should benefit more than the sequential domain (61.5% routing). The data in Table 23, however, shows little correlation between area reduction and the sizes of the routing fabrics in the architectures, which we saw in Table 21.


```

findSwitchCount(n, m)
{
    //min and max possible switch counts for binary search
    min = n;
    max = n*m;

    swCount = (max+min)/2; //current switch count we're attempting

    while(swCount != min && swCount != max) {
        sw = switchPlacementAlgorithm(n, m, swCount); //create crossbar

        //attempt to route, adjust swCount according to success/failure
        if(runPathfinder(circuits, sw)) {
            max = swCount;
            swCount = (min+swCount)/2;
        } else {
            min = swCount;
            swCount = (max+swCount)/2;
        }
    }

    return sw[n][m];
}

```

Figure 90. Pseudocode for using a binary search to find the smallest sparse crossbar that the CPLD successfully routes on

The reason that the sparse crossbars are providing similar area improvements is that the sparseness of the crossbars is varying according to the size of the CPLD. The circuits in the sequential domain have fewer signals to route, and they can therefore subsist on very sparse crossbars. The circuits in the encryption domain, however, have a large number of signals to route, and require more connectivity in their crossbars. Table 24 displays this phenomenon, listing the crossbar switch densities of the different domains and relating it to the sizes of the respective CPLD architectures using these sparse crossbars.

Table 23. Performance comparison between full-crossbar results and sparse-crossbar results for the architectures found in the last chapter

Domain	Full Crossbar				Sparse Crossbar			
	Arch	Area	Delay	A*D	Arch	Area	Delay	A*D
Sequential	14-38-4	1.00	1.00	1.00	14-38-4	0.48	0.44	0.21
Arithmetic	10-22-2	1.00	1.00	1.00	10-22-2	0.37	0.35	0.13
Combinational	12-70-4	1.00	1.00	1.00	12-70-4	0.33	0.37	0.12
Floating Point	8-18-2	1.00	1.00	1.00	8-18-2	0.40	0.41	0.16
Encryption	8-32-2	1.00	1.00	1.00	8-32-2	0.46	0.47	0.22
Geo. Mean		1.00	1.00	1.00		0.40	0.40	0.16

Table 24. Switch densities of sparse crossbars related to the area of the sparse-crossbar-based CPLD

Domain	Normalized Area	Switch Density
Sequential	1.00	13.6%
Combinational	15.19	20.0%
Arithmetic	16.19	28.8%
Floating Point	81.66	35.5%
Encryption	961.32	41.8%

Now that we are using sparse crossbars, with augmented area and delay models to account for their effects, we might expect the Architecture Generator to find different architectures than those it found for CPLDs with full crossbars. This is exactly what happened, as the new models led the Architecture Generator to find a new architecture for each of the domains that we are considering. Table 25 displays the architecture results obtained by our tool flow using the new area and delay models, and compares the results to those found in the previous chapter. The best sparse-crossbar-based CPLDs are shown to require .37x the area, .30x the delay, and .11x the area-delay product of our best full-crossbar-based CPLDs.

Table 25. Best results found for full and sparse-crossbar-based CPLDs

Domain	Full Crossbar				Sparse Crossbar			
	Arch	Area	Delay	A*D	Arch	Area	Delay	A*D
Sequential	14-38-4	1.00	1.00	1.00	14-18-4	0.36	0.48	0.17
Arithmetic	10-22-2	1.00	1.00	1.00	3-16-2	0.31	0.25	0.08
Combinational	12-70-4	1.00	1.00	1.00	14-52-3	0.40	0.39	0.15
Floating Point	8-18-2	1.00	1.00	1.00	18-55-3	0.37	0.21	0.08
Encryption	8-32-2	1.00	1.00	1.00	23-79-4	0.41	0.27	0.11
Geo. Mean		1.00	1.00	1.00		0.37	0.30	0.11

8.5.1 Using Other Evaluation Metrics

As with our full-crossbar based CPLD flow, we also ran our sparse-crossbar based CPLD flow in both area driven and delay driven modes. The results of this are shown in Table 26, which displays the results of running our chosen algorithm on each of our five domains for area-delay driven, area driven, and delay driven metrics. All results are again normalized to the area-delay driven results.

Table 26. Results of running our chosen algorithm in area-delay driven, area driven, and delay driven modes for sparse-crossbar based CPLDs

Sparse XBAR	Area*Delay Driven			Area Driven			Delay Driven		
	Area	Delay	A*D	Area	Delay	A*D	Area	Delay	A*D
comb	1.00	1.00	1.00	0.80	0.93	0.75	1.13	0.81	0.91
seq	1.00	1.00	1.00	1.07	1.17	1.25	1.49	0.83	1.24
fp	1.00	1.00	1.00	0.92	1.74	1.60	1.52	0.68	1.04
arith	1.00	1.00	1.00	0.93	1.18	1.10	3.39	0.61	2.08
enc	1.00	1.00	1.00	1.04	1.43	1.50	1.60	0.48	0.77
GeoMean	1.00	1.00	1.00	0.95	1.26	1.20	1.69	0.67	1.13

Table 26 shows that, when creating sparse-crossbar based CPLD, both area and delay driven modes are successful at optimizing for their target metric. Using area as our metric, our domain-specific architectures require only .95x the area of those found with area-delay as the metric, and using delay as the metric we find architectures that require only .67x the delay of those found with area-delay as the metric.

The analysis of Table 26 is very similar to that of Table 20, which showed results for full-crossbar based CPLDs. Using delay as our metric, we are able to find architectures where PLAmapping was very successful at optimizing for delay (.67x of area-delay driven), but not successful at optimizing for area (1.69x of area-delay driven). The area driven results are again very close to the area-delay driven results, requiring .95x the area and 1.26x the delay of the architectures found using area-delay as the metric. Basically, when PLAmapping is very successful at optimizing for delay, it causes the area minimizing heuristics to be less successful. These are the architectures found by delay driven mode. Conversely, when PLAmapping is not as successful at delay optimization, the area minimizing heuristics are more successful. These are the architectures found by area driven mode.

An interesting side note from Table 26 is that three of our area and delay driven architectures actually provide better area-delay performance than our area-delay driven architectures. This is an artifact of the fact that we must route our circuits on these sparse-crossbar based CPLDs architectures, determining the minimum switch count on which the circuits will route. The points that display improved area-delay performance are instances where the router was somewhat lucky and successfully routed all the circuits on a small sparse crossbar, therefore providing better delay and area performance than was predicted by our models.

8.6 Conclusions

This chapter introduced the use of sparse crossbars in our CPLD architectures, including the methods used to create the sparse crossbars for a specific architecture. Sparse crossbars decrease the area and capacitance of the CPLD interconnect structure, providing area and delay gains over full crossbars. Sparse crossbars do not provide full capacity, however, so a routing algorithm was introduced to ensure that the CPLD's signals can all be routed.

Incorporating sparse crossbars into the architectures found in the previous chapter resulted in architectures that required .40x the area and .40x the delay of those with full crossbars. Running the Architecture Generator with new area and delay models was able to find even better architectures, requiring .37x the area and .30x the delay of those found in the previous chapter. The performance gains were evenly spread across the domains, despite the fact that larger CPLDs seem to have more to gain from reducing the routing area. This was explained by the fact that domains that require more routing resources are not able to depopulate their crossbars as much as domains that require fewer routing resources.

Also, using area and delay driven modes, we were able to find architectures that were more optimized for their respective metrics. Architectures found in area driven mode required .95x the area and 1.26x the delay of those found using area-delay, and

architectures found in delay driven mode required .67x the delay and 1.69x the area of those found using area-delay as the metric.

9 Adding Capacity to Domain-Specific CPLDs

We envision that many SoC designers who use our flow will know the circuits that they wish to implement in reconfigurable logic on their device. They can simply provide us with the circuits, and our automated flow will create an architecture that is guaranteed to efficiently implement their circuits.

Some designers, however, will not necessarily know the circuits they are going to implement in reconfigurable logic. Perhaps the design of the whole SoC is still being finalized, and they are unsure of what portions will be implemented in reconfigurable logic. Another possibility is that the reconfigurable fabric will be used to implement some common protocol that is going to be updated or modified shortly, and the fabric will need to support the new version. Along a similar vein, what happens if a bug is found in the circuit that is being implemented in reconfigurable logic, and it needs to be slightly modified?

In these examples, the exact circuits that the reconfigurable logic will implement cannot be provided in the testing set. The designer will likely know the general domain of these circuits, however, and could therefore provide us with circuits that are similar to those that the fabric must support. The question then becomes: given a domain of circuits, how can we create an architecture that not only supports the sample circuits, but which is as likely as possible to support an unknown circuit in the same domain.

The general solution to this problem is to take the base architecture created by the tool flow, and to augment it with resources that are similar to what it already includes. The base architecture will contain a good sample of the resources that the domain requires, so adding more of these resources should help it support unknown, future circuits. But is this actually the correct answer, and what exactly does “more of the same

resources” mean? Specifically, what resources should be added, and in exactly what resource mixture? This is the question that is explored in this chapter.

9.1 Adding Capacity to CPLDs

In order to add “capacity” to our CPLDs, we’ll first need to identify the CPLD architectural characteristics that can be easily augmented. In terms of routing resources, the complete network implementation of our CPLDs guarantees that every signal is available to every PLA in the architecture, so simply adding more routing tracks is unlikely to be beneficial. We are using sparse crossbars to connect to the PLAs, however, so the switch density of the crossbars is something that can be modified in order to support routing rich designs. In terms of logic resources, our CPLDs exclusively use PLAs. PLAs can be modified in terms of their input, product-term, and output counts, and we can also modify how many PLAs we have in our architecture. This gives us a total of five variables to consider when adding capacity: crossbar switch density, PLA input count, PLA product-term count, PLA output count, and PLA count.

Another thing to note is that the input, product-term, and output counts of PLAs are related. While it might be beneficial to augment only one of these items in the PLAs in our architecture, we are more likely to see benefits if we augment all three of the variables in some intelligent fashion. We have already determined the in-pt-out ratio that the architectures desire, so our first strategy will be to augment all three variables using their existing in-pt-out ratio. For a second strategy, we will simultaneously augment each of the variables by an additive factor.

Lastly, there might be some utility to a strategy that uses larger PLAs AND allows an increase in PLA count: we will consider this strategy as well, using a fixed multiplicative factor to make the PLAs larger. Table 27 lists the CPLD augmentation strategies that we will employ in this section.

For the strategies which have fixed PLA sizes but which allow additional PLAs, we find our mapping simply by providing PLAMap with the PLA size of the architecture. PLAMap then provides us with the required PLA count, and the performance

characteristics are calculated. For strategies that alter the PLA sizes while keeping the PLA count fixed, we iteratively call PLAMap with larger and larger PLA sizes (using the smallest possible increment) until a mapping fits the PLA count constraint. If the PLA variable(s) in question get increased to three times their initial values without finding a mapping that fits the PLA count constraint, then it is considered a failure.

Table 27. Strategies for adding capacity to our CPLD architectures

Strategy	Notes
Switch Density	Augment the switch density of sparse crossbars by x%, allow extra PLAs.
PLA Count	Allow extra PLAs of the base size.
PLA Inputs	Augment the input count of each PLA, but keep PLA count fixed.
PLA Product Terms	Augment the pterm count of each PLA, but keep PLA count fixed.
PLA Outputs	Augment the output count of each PLA, but keep PLA count fixed.
PLA Size1	PLA size = $(c \cdot in, c \cdot pt, c \cdot out)$, but keep PLA count fixed.
PLA Size2	PLA size = $(d + in, d + pt, d + out)$, but keep PLA count fixed.
PLA Size1 + PLA Count	PLA size = $(c \cdot in, c \cdot pt, c \cdot out)$, and allow extra PLAs of this size.

9.2 Methodology

We want to simulate a situation in which the SoC designer knows the general domain of circuits that will be implemented in reconfigurable logic, but does not necessarily know the exact circuits. We accomplish this by taking the domains that we already have and by removing one or more of the circuits in order to create reduced domains. These reduced domains are used to create domain-specific architectures according to our tool flow. If the architecture created for the reduced domain is different than the architecture for the full domain, then we reintroduce the removed circuit(s) and determine what additional CPLD resources (if any) are required to implement the new circuit.

This process was applied to every circuit in each of our five domains. Each circuit was individually removed, and the reduced architecture was created. We also created five new sub-domains by grouping very similarly sized circuits from the main domains, and we applied this process to each circuit in these sub-domains as well. These ten domains had a total of 92 circuits in them.

Each of the 92 circuits was removed from its domain, and 35 of the corresponding reduced architectures were different than their full-domain architectures. Using

knowledge of the 35 cases where removing a single circuit resulted in a different architecture, we then created 14 more such cases by removing multiple circuits from the domains. In all, we created 49 reduced architectures that were different from the architectures obtained by the full domains.

With these 49 interesting reduced architectures, we will be intelligently adding resources to the architectures in order to determine what is required to support the removed circuits. The basic question we wish to answer is this: given an architecture augmentation strategy, how much of an area penalty must we tolerate in order to support the additional circuits? We will analyze this information using graphs of the form shown in Figure 91. In the figure, the x-axis displays the area requirement of a given strategy, normalized to the reduced domain. The y-axis displays the number of domains that can support their additional circuits for the specified area requirement. Results that are toward the upper left of the graph are good, because many domains can be supported by the strategy with a small area overhead. Results that are toward the lower right of the graph are bad, because few domains are supported and the area penalty is high.

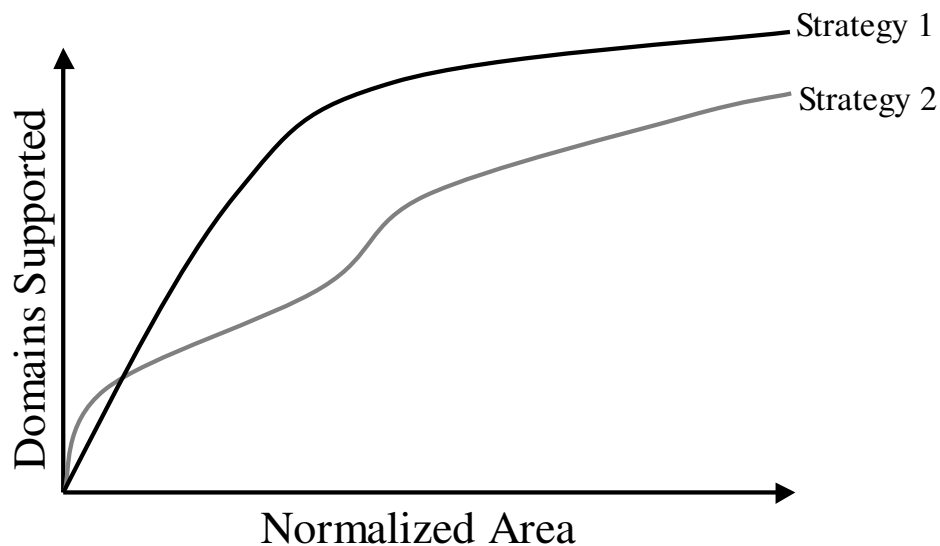


Figure 91. The basic graph we will use to examine our different CPLD augmentation strategies

We will also compare the different resource strategies according to the geometric mean of their data points. Each strategy will have data points in the exact same y-

locations on the graphs, so their x-values will be normalized to a particular strategy, and their means calculated and compared: this will give a numerical value that can be used for comparisons. The graphical data can be difficult to compare at times, and the graphs often do not include all the data points (in order to increase the visibility of the interesting areas), so a numerical metric will be helpful.

9.2.1 Routing Failures

When a circuit is mapped such that it fails to route on an architecture, there are two possible ways to get the circuit to route: we can spread the mapping out among additional PLAs, or we can make the existing PLAs larger. Both of these strategies will eventually allow a circuit to route, but they have very different effects on the architecture to which we are mapping.

Several of our augmentation strategies increase the number of PLAs in our architecture until the reintroduced circuit(s) fit on the architecture. Under these strategies, the PLAs are set to a specific size and cannot be augmented. If a circuit fails to route under this augmentation strategy, therefore, the only option is to spread the mapping out among more PLAs. We spread out the mapping by running PLAmapping on the circuit, but with smaller PLAs than our actual architecture will use. For example, if a circuit failed to route on an architecture using 10-20-5 PLAs, we will then run the circuit through PLAmapping with a 9-20-5 architecture, acquire the mapping, and attempt to route it on the 10-20-5 architecture. If it fails to route again, a mapping using 8-20-5 PLAs will be acquired, and we will again attempt to route it on the 10-20-5 architecture. The next attempt would use a 10-20-4 mapping, then a 9-20-4 mapping, then 8-20-4, then 7-20-5, and so on, until the circuit successfully routes. This algorithm is shown in Figure 92, including the method for deciding upon the next mapping to attempt.

After compiling the data for this section, we determined that we should have been using the strategy of spreading out mappings for ALL of the cases where we see a routing failure. Routing failures occurred in so few cases, however, that we decided not to retake all the data with this routing-failure strategy implemented. Rather, we will provide

graphs that show the best possible performance that could have been achieved from implementing this routing-failure strategy, and we will display that the conclusions drawn from the figures are unaffected by this omission. Figure 94 is actually the only graph that uses augmentation strategies in which the PLA count of the architecture is fixed, so it is the only figure affected.

```

routeBySpreadingMapping()
{
    input circuit;          //the circuit
    input swPerc;          //the switch percentage
    input in, pt, out;     //base PLA size is in-pt-out
    in2, out2;            //reduced PLA variables for getting mappings
    ratio = ceil(in/out) //the ratio of inputs to outputs, rounded up
    s, t;                 //integers used in the loop

    //Loops and variables to control what we set the in2 and out2
    //variables to be. We will then get a mapping for a in2-pt-out2
    //architecture and route it on an in-pt-out architecture
    for(i=1; i<in; i++) {
        s = i-1;
        t = -1;
        for(j=0; j<(ratio*i); j++) {
            if(t++ == ratio) {
                s--;
                t = 1;
            }
            in2 = in-j;
            out2 = out-s;
            //if it successfully routes at this mapping, we're done
            if(runPLAmapAndRoute(circuit, swPerc, in2, pt, out2, in,pt,out)) {
                return {in2, pt, out2};
            }
        }
    }
    throwError(); //should always route at SOME mapping size
}

```

Figure 92. Pseudocode for spreading out the mapping of a circuit among additional PLAs in order to achieve a legal route

Notice that we are only decreasing the input and output values of the mapping, not the product-term value. This is because decreasing the input and output values is going to have a greater effect on the routability of our mapping, allowing us to converge to a routable solution more quickly. Our routing problem involves routing signals from PLA outputs to interconnect tracks to PLA inputs, so if our mappings use fewer inputs or outputs, it will decongest the routing in these regions. Decreasing the number of product terms in a mapping might spread the mapping out among more PLAs, but it would do little to decongest the routing resources at the PLA inputs and outputs, and would therefore be less effective.

For all of the augmentation strategies which allow PLAs to increase in size, no special intervention is used to facilitate routing. In these cases, if a mapping fits the constraint of PLA count but fails to route, then it is considered a failure (just as if it had not met the PLA count constraint). The algorithm will continue to increase the PLA size until a corresponding mapping routes on the architecture.

9.3 Results

When mapping to reconfigurable architectures, providing sufficient routing resources is necessary if one wishes to fully utilize the architecture's logic elements. If insufficient routing resources are available, mappings either fail to succeed or they get spread out so that they require more logic elements than inherently necessary. This is true of our CPLD architectures, as providing insufficient switch density in the crossbars will either make architectures unroutable or it will force the PLAs to be underutilized, causing an increase in the number of PLAs required. This suggests that finding a good crossbar switch density should be the first thing done in our architectures, as providing sufficient crossbar connectivity will allow any increased logic resources to be utilized efficiently.

Our architectures are initially given the minimal switch density that allows all the circuits in a domain to map, but this might be insufficient for future circuits. We therefore ran an exploration of how switch density affects the amount of area required to

map future circuits to our reduced architectures. In this exploration, the reduced architectures were given additional switches according to a multiplicative factor and allowed to use as many extra PLAs as necessary in order to map the additional circuits. The results of this exploration are shown in Table 28 and Figure 93, and the data represents architectures with the *base* number of switches, *base**1.05 switches, *base**1.10 switches, *base**1.20 switches, and *base**1.50 switches.

The data shows that good results are obtained when either 0% or 5% extra switches are added to the basic crossbar in the CPLD. Indeed, for any normalized area shown in the graph, either the $c=1.00$ or the $c=1.05$ graph provides the most domains supported. Also note that the $c=1.05$ graph always outperforms the $c=1.10$, $c=1.20$, and $c=1.50$ graphs, so there appears to be no benefit from adding more than 5% to the crossbar switch density.

But which strategy is better, using the base crossbar switch density or adding 5%? According to Table 28 it is somewhat of a wash. If a new circuit can be mapped using 0% additional switches, then an architecture with 5% additional switches will end up requiring up to 5% more area than the architecture with no additional switches. If a new circuit successfully maps to an architecture with 5% more switches but fails on an architecture with no additional switches, however, then the mapping will have to be spread out among additional PLAs in order to reduce the routing demands. In this case, the area cost might be much more than 5%, as it depends on how many additional PLAs are required when the mapping gets spread out. Adding 5% to the crossbar switch density therefore provides better worst-case behavior than not adding any switches. This is one reason why we have concluded that one should add 5% to the crossbar switch density when trying to support unknown circuits.

Table 28. Results of attempting to support future circuits by adding switches to our crossbars and allowing any number of PLAs in the architecture. Results are geometric mean

Switch Count	Mean Result
BASE	1.00
BASE * 1.05	1.01
BASE * 1.10	1.03
BASE * 1.20	1.07
BASE * 1.50	1.18

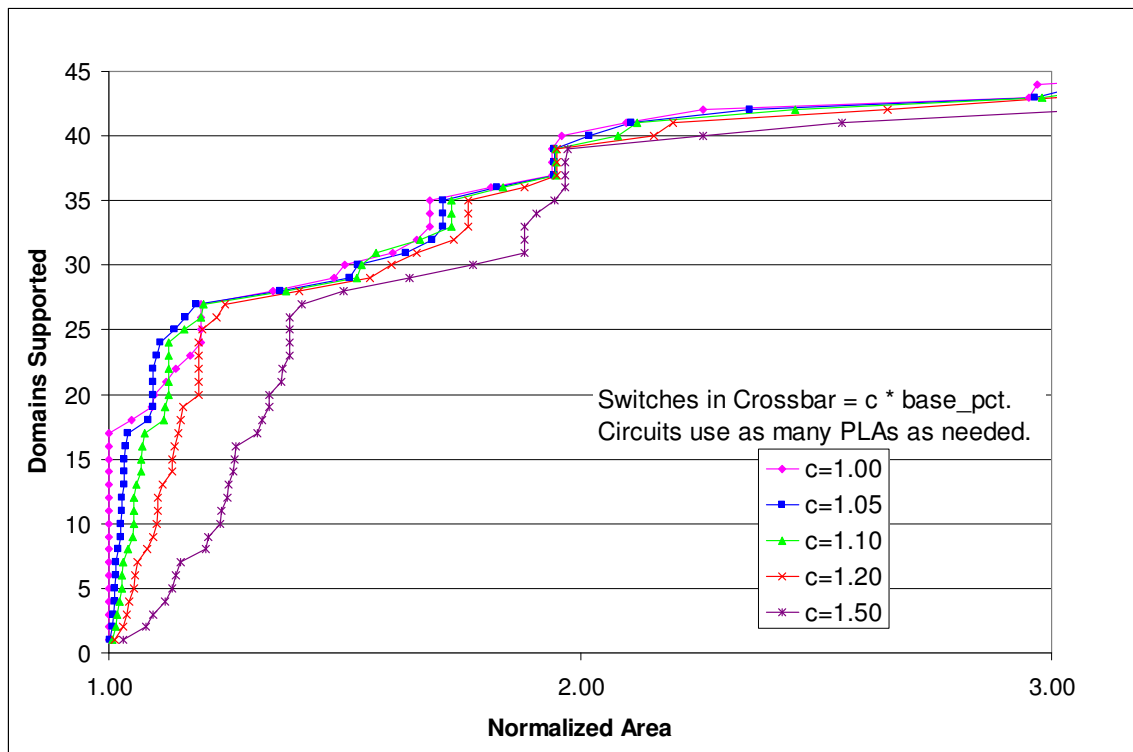


Figure 93. Results of attempting to support future circuits by adding switches to our crossbars and allowing any number of PLAs in the architecture

Another reason is shown in Table 24 from the previous chapter. This table showed that larger architectures required a higher switch density than smaller architectures, so it makes sense that we should provide some additional switch density if we are going to be making our architectures larger. For all future explorations in this section, therefore, we will be adding 5% to the base switch density in the corresponding architectures.

Having dealt with the routing structure, we must now examine the logic resources. We can augment the logic in our reduced architectures by adding PLAs, by adding inputs, product terms, or outputs to our PLAs, or by adding all three of these variables to our

PLAs in either a multiplicative manner, $c^*(in-p-out)$, or in an additive manner, $c+(in-pt-out)$. In the strategies in which we increase the sizes of the PLAs, the architecture is not permitted to use more PLAs than the base architecture. We performed each of these experiments with our 49 interesting reduced architectures, and acquired the results shown in Table 29 and Figure 94.

Table 29. Results of attempting to support future circuits by adding the specified logic resources. All strategies use 5% more switches than the base reduced architecture. Results are geometric mean, and failure rate details how many domains were not supported by a strategy

Strategy	Mean Result	Failure Rate
#PLAs	1.00	0%
#PLA inputs	1.05	51%
#PLA pterms	1.02	49%
#PLA outputs	1.02	49%
$c^*(in-pt-out)$	1.16	0%
$c+(in-pt-out)$	1.24	0%

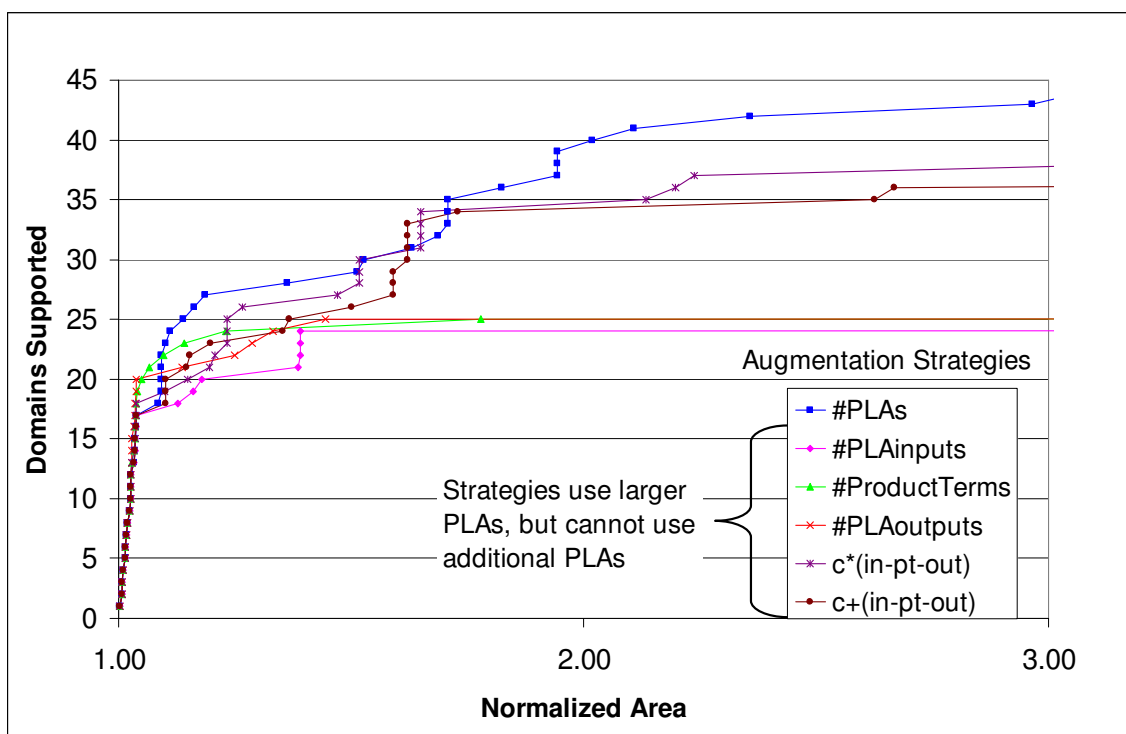


Figure 94. Results of attempting to support future circuits by adding the specified logic resources. All strategies use 5% more switches than the base reduced architecture

As the data shows, the most efficient strategy for supporting additional circuits is simply to add more PLAs to the architecture. The multiplicative and additive strategies

are also capable of supporting all of the additional circuits, but they do not do it as efficiently as the strategy of adding more PLAs. Of the multiplicative and additive strategies, the multiplicative strategy appears to perform slightly better for most of the points in the graph, and this is also shown in the table. The strategies in which we simply add inputs, outputs, or product terms to the PLAs are insufficient to even support all the additional circuits, as the logic resources end up lacking in the variables that don't get augmented and they are unable to support many of the circuits.

As discussed above, some of the augmentation strategies shown in Figure 94 do not use a routing-failure recovery strategy. This includes all but the blue data (#PLAs) in Figure 94. Table 30 and Figure 95 show the best possible results that could have been achieved by these strategies had they been using the routing-failure recovery strategy described above. As can be seen, the differences between Figure 94 and Figure 95 are practically imperceptible, and the same analysis that we applied to Figure 94 can be applied to Figure 95. The similarities between Table 29 and Table 30 also back up this conclusion.

Table 30. The best case performance of the data in Table 29, assuming a routing-failure recovery strategy had been used

Strategy	Mean Result	Failure Rate
#PLAs	1.00	0%
#PLA inputs	1.03	51%
#PLA pterms	1.01	47%
#PLA outputs	1.01	49%
c*(in-pt-out)	1.14	0%
c+(in-pt-out)	1.22	0%

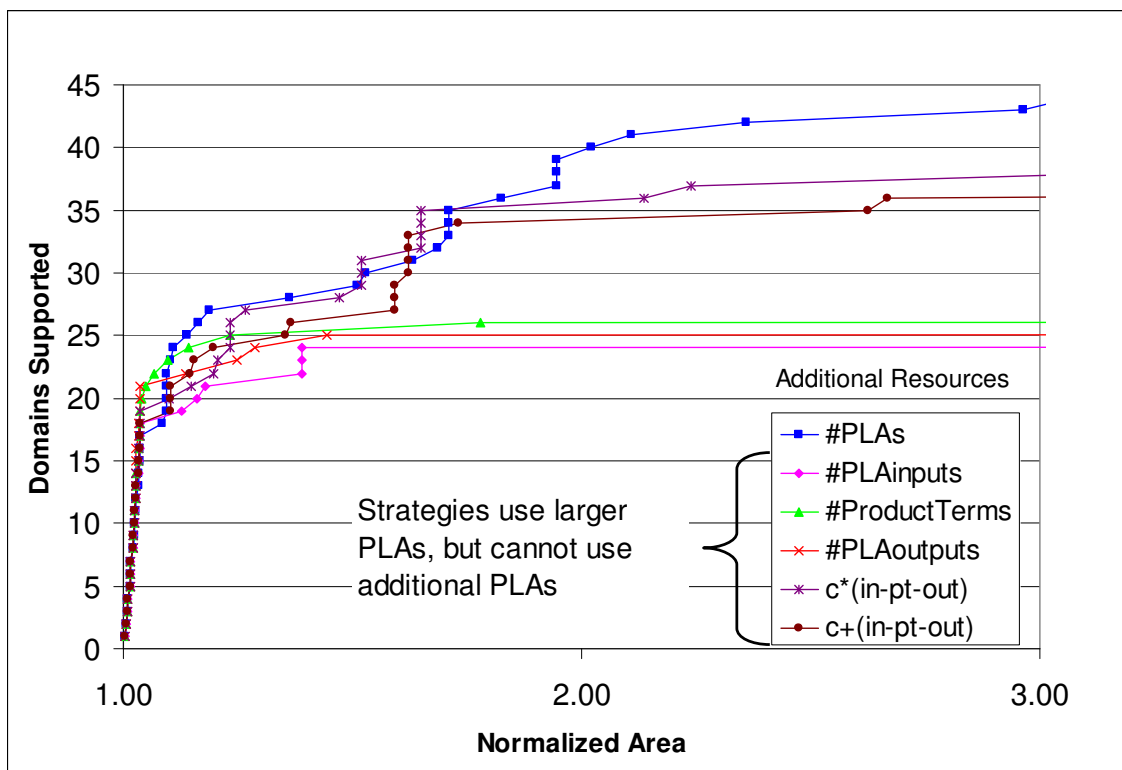


Figure 95. The best case performance of the data in Figure 94, assuming a routing-failure recovery strategy had been used

The strategies of adding PLAs and adding inputs/outputs/product terms to the PLAs multiplicatively are both shown to perform reasonably well in Figure 94, so another idea is to attempt a mixture of these two strategies. The idea here is to make the PLAs some percent larger, and then to allow the implementation to use as many PLAs as it requires. Table 31 and Figure 96 show the results of doing this with PLAs that are 0%, 10%, 20%, 50%, and 100% larger than the base size.

Table 31. Results of attempting to support future circuits by augmenting the PLA size and allowing any number of PLAs. All strategies use 5% more switches than the base reduced architecture

PLA Size	Mean Result
BASE	1.00
BASE * 1.10	1.10
BASE * 1.20	1.23
BASE * 1.50	1.79
BASE * 2.00	2.79

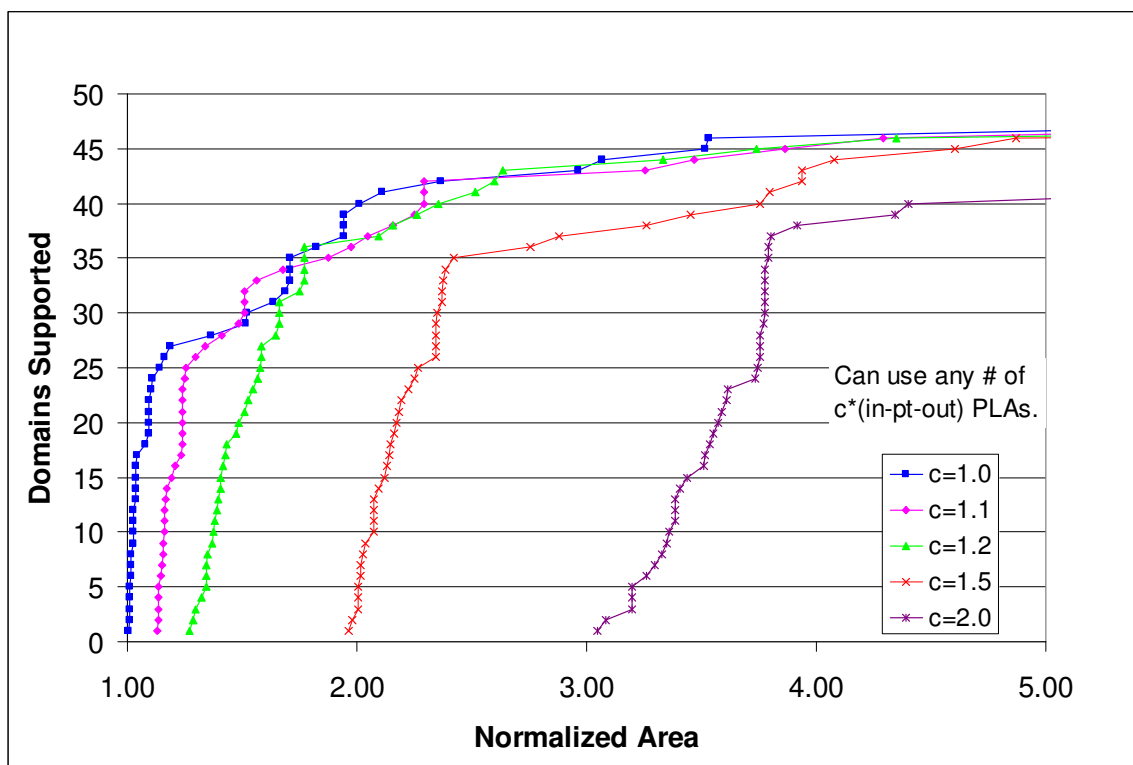


Figure 96. Results of attempting to support future circuits by augmenting the PLA size and allowing any number of PLAs. All strategies use 5% more switches than the base reduced architecture

As the data displays, adding PLAs of the base size still appears to be our best augmentation strategy. Making the PLAs 10% or 20% larger also provided reasonable results, but adding any more than 20% is shown to be area-prohibitive. These results are much worse than we expected, as we thought that a hybrid strategy would perform comparably to the strategy of simply adding more PLAs of the base size. Closer examination of this particular hybrid strategy showed us that we had approached the problem in the wrong way.

This hybrid strategies begins by augmenting the PLAs in the architecture by a fixed percentage, and this has the obvious effect of making the architecture larger. Using larger PLAs may allow future circuits to require fewer PLAs, but since we are only adding resources (not removing), we cannot reduce the number of PLAs in the architecture below the number in the base architecture. Because of this, the minimum area achievable by this hybrid strategy is going to be larger than the strategy which does

not make the PLAs larger. This can be seen in Figure 96, as the lines are no longer intercepting the x-axis at a normalized area of 1.00. The basic pitfall of this hybrid strategy is that we are not adding resources incrementally: we are instead providing a large initial boost to the resources in terms of PLA size, and then allowing incremental changes from that point on (in terms of additional PLAs).

We thus developed a second hybrid strategy that is capable of adding resources incrementally. In this new strategy, λ of the additional area resources are provided to additional PLAs, while $(1 - \lambda)$ of the additional area resources are provided to larger PLAs (using a multiplicative scaling factor), where $0 \leq \lambda \leq 1$. Resources are slowly added, using these ratios, until the removed circuit is supported by the architecture. Table 32 and Figure 97 display the results of running this new hybrid strategy with λ values of 1.00, 0.75, 0.50, 0.25, and 0.00.

This new hybrid method is successful at adding resources in an incremental fashion, as shown by the fact that the lines in Figure 97 are intersecting the x-axis at a normalized area of 1.00. The strategies of giving 75% and 50% of the additional area resources toward more PLAs are both shown to be relatively effective, but the data still demonstrates that the most effective method of supporting future circuits is simply to add additional PLAs of the base size.

Table 32. Results of using the new hybrid strategy. All strategies use 5% more switches than the base reduced architecture, and can use as many PLAs as required

λ	Mean Result
1.00	1.00
0.75	1.05
0.50	1.05
0.25	1.17
0.00	1.16

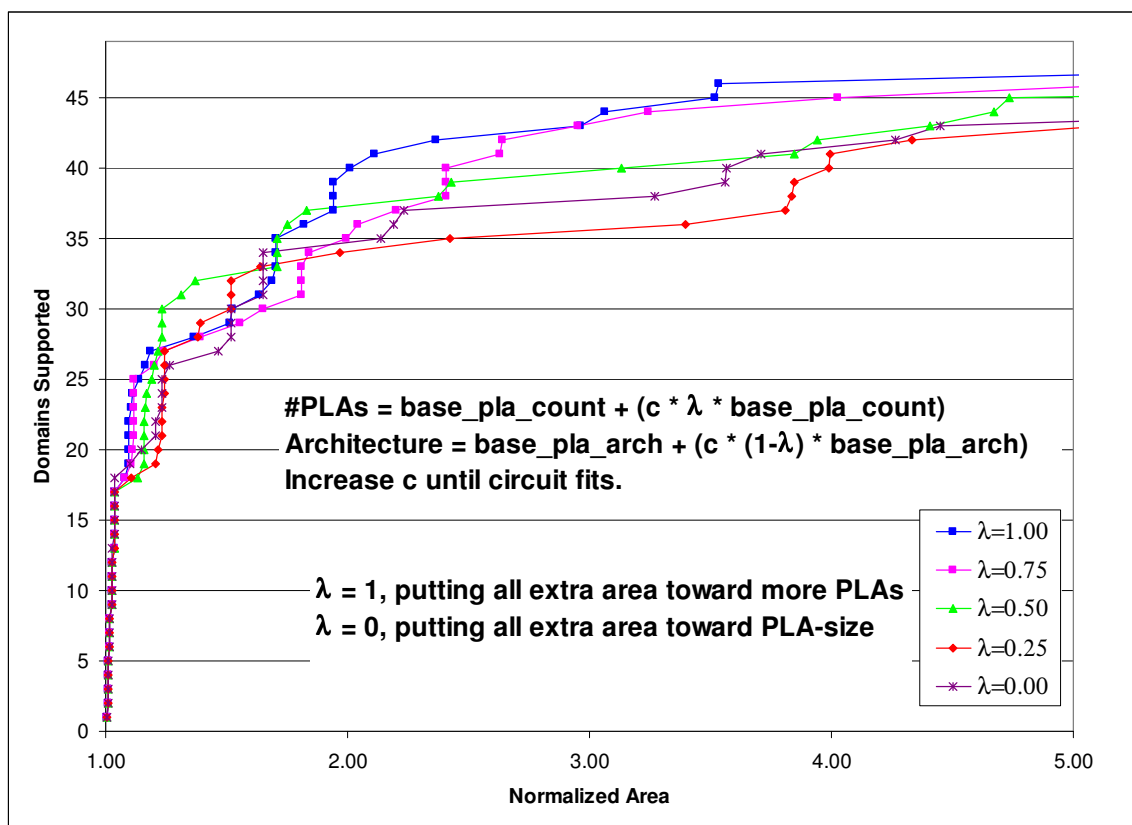


Figure 97. Results of using the new hybrid strategy. All strategies use 5% more switches than the base reduced architecture, and can use as many PLAs as required

9.4 Conclusions

In this chapter we explored the concept of adding capacity to our domain-specific CPLDs. Some SoC designs will require their reconfigurable logic to support circuits that have not yet been specified, so we need to examine the best way to add resources to our architectures such that we maximize the likelihood of future circuits fitting on the architecture.

Our results showed that, in terms of routing resources, it is good to add 5% to the switch density of the sparse crossbars used in the CPLD architectures. In terms of logic resources, the most area-efficient method of supporting future circuits is simply to add more PLAs of the same size found in the base architecture. This is consistent with the strategy that is most commonly employed with reconfigurable devices, in which

additional capacity is provided by adding more of the resources that already exist in the architecture.

10 Conclusions and Future Work

Reconfigurable logic fills a useful niche between the flexibility provided by a processor and the performance provided by custom hardware. This usefulness extends to the SoC realm, where reconfigurable logic can provide cost-free upgradeability, conformity to different but similar protocols, coprocessing hardware, and uncommitted testing resources. Additionally, the paradigm of IP reuse makes it easy to incorporate reconfigurable logic into an SoC device, because it can be provided to the designer as a pre-made IP core.

General reconfigurable logic suffers performance penalties due to its flexible nature, as it must be capable of supporting a wide range of designs. A unique opportunity exists when creating reconfigurable fabrics for SoC, because the designer will already know the application domain that the device will be targeting. Using this information, a domain-specific reconfigurable fabric can be designed that will target the exact applications that it will need to support on the SoC, providing improved performance over more general fabrics. The dilemma then becomes creating these domain-specific reconfigurable fabrics in a short enough time that they can be useful to SoC designers. The Totem project is our attempt to reduce the amount of effort and time that goes into the process of creating domain-specific reconfigurable fabrics, thereby providing SoC designers with efficient reconfigurable architectures without adversely affecting their design schedules.

10.1 Contributions

This dissertation presented processes for creating domain-specific product-term-based reconfigurable architectures for use in SoC devices. This included a complete flow for creating domain-specific PLAs and PALs: a project that we call Totem-PLA. Also included was a complete flow for creating both full- and sparse-crossbar based CPLD

architectures: a project termed Totem-CPLD. In addition to providing methods for creating these architectures, we have also provided guidelines as to how to augment our CPLD architectures with additional resources in order to maximize the likelihood that they will support future, unknown circuits.

In Chapter 6 we introduced a complete flow that can be used to create domain-specific PLA and PAL devices for use in SoC applications. By intelligently mapping circuits to these devices using simulated annealing, up to 73% of the programmable connections in the AND and OR-planes could be removed while still supporting the circuits. This led to delay improvements of 16% to 31% in the arrays, although compaction was unable to improve upon the area of our automatically generated layouts.

Chapter 7 next introduced a complete flow that can be used to create domain-specific CPLD architectures that are based on full crossbars. We presented several algorithms that, given a domain of circuits as input, are capable of finding a PLA size that results in a CPLD architecture that efficiently supports the given circuits. Of the algorithms, the Run M Points algorithm with a second iteration was shown to provide the best results. This algorithm was able to create domain-specific architectures that outperform representative fixed CPLD architectures by 5.6x to 11.9x in terms of area-delay product. Even the best fixed-architectures that we could identify still performed 1.8x to 2.5x worse than our domain-specific architectures. This flow also included a Layout Generator that creates a full VLSI layout of the specified CPLD architecture in the tsmc .18 μ process.

One of the main drawbacks of CPLDs is that their interconnect structures grow quadratically with respect to PLA count. In order to help alleviate these area concerns, Chapter 8 introduced the concept of using sparse crossbars in our CPLD architectures. By utilizing an existing sparse-crossbar generation tool and implementing a congestion-based router, we were able to create CPLD architectures based on sparse crossbars that required only 37% of the area and 30% of the propagation delay of their full-crossbar based counterparts. A smoothing algorithm was also introduced which allowed us to distribute switches evenly throughout the sparse crossbars: this allowed us to create layouts that were as compact as possible.

Finally, Chapter 9 addressed the question of how to add additional resources to a sparse-crossbar based CPLD in order to maximize the likelihood that unknown circuits will be supported by the architecture. Results showed that the sparse crossbars should be augmented with 5% more switches than were required in the base architecture, and that the best way to add logic to the architecture is to add more PLAs of the base size.

Taken as a whole, this work provides a framework for the creation of domain-specific product-term-based reconfigurable architectures for use in SoC devices. We have introduced algorithms that effectively create PLA, PAL, and CPLD architectures, as well as providing tools that create high-performance VLSI layouts from these architecture specifications. We have also explored the question of adding capacity to these CPLD architectures, and provided guidelines for which resources should be added in order to maximize the likelihood of supporting future designs.

While our architecture generation tools have used practical methods and provided quality results, the tools described in this dissertation are in no way meant to be production quality. Rather, they are intended to provide a framework for the creation of domain-specific architectures, particularly those that are based on product-term style logic. The methods employed in this work can be seen as guidelines for how to undergo the different processes involved in creating domain-specific architectures.

10.2 Conclusions and Future Work

The work on Totem-PLA was intended to give us insight into how we can tailor PLAs to a specific application domain. The eventual goal was to apply this knowledge to the creation of PLA-based CPLDs, allowing us to tailor both the high-level CPLD architecture and the low-level PLA architecture to the target domain. While we were able to provide some delay gains by depopulating the AND- and OR-planes of the PLA and PAL devices, we determined that this would come at too high of a cost to flexibility. Therefore, when creating domain-specific CPLD architectures, we were only able to modify the PLAs according to their input, product term, and output sizes.

Attempts to compact our depopulated PLAs and PALs provided us with no area gains. In hindsight, it is apparent that we approached this problem in the wrong manner. PLAs and PALs are such regular structures that, even if you depopulate their arrays by more than 50% (which we did), a compaction tool would not be expected to provide any real benefits. We should have designed the PLAs and PALs with the intent of leveraging the packing methods that we employed for the sparse crossbars in Chapter 8. The layouts of the switches in the sparse crossbars are very similar to the layouts of the AND- and OR-plane cells in the PLAs and PALs, and the exact same methods could have been employed in order to pack the cells in these planes. The only additional consideration would be the drivers, pull-up devices, and registers at the peripheries of the AND- and OR-planes. These cells could have been designed and laid out with multiple possible aspect ratios such that, after the dimensions of the AND- and OR-planes are determined, it is just a matter of choosing the periphery cells with the proper aspect ratios. If any future work on PLA or PAL array depopulation is undertaken, such a strategy should be implemented.

When sizing the devices in our final layouts, we attempted to choose sizes that would result in reasonable performance for a wide range of PLA, PAL, and CPLD architectures. The basic idea was to trade a small amount of architecture performance for lower design complexity. Unfortunately, the architectures we create can vary in size by more than three orders of magnitude. Considering the wide variation we see in architecture size, it is clear that the use of fixed device sizes must be producing significant penalties in our delay performance. An obvious avenue of future work would be to provide devices with multiple sizing options, such that the devices in the final layout are sized according to the characteristics of the final architecture. While we have not performed an analysis of the performance gains this would provide, our guess is that delay improvements on the order of 10x may be achievable for larger architectures.

Another concept that we did not consider is power dissipation. The standard metric that we used to evaluate our architectures was area-delay product, but we also looked at the architectures that would be chosen when using simple area-driven or delay-driven

flows. An obvious avenue of future work would be to create a power model for our architectures using a tool like Powermill, and to run our tool flow using cost functions that integrate power into the equation. This would likely lead to architectures that penalize PLAs for their product terms and outputs, since the number of power hungry pseudo-nMOS gates in one of our PLAs equals the sum of the product term and output counts.

It would also be desirable to create architectures that don't use pseudo-nMOS gates at all, since gates of this style tend to dissipate a large amount of static power. While most commercial CPLDs still use pseudo-nMOS PLAs and PALs, Xilinx has developed a low power CoolRunner CPLD family [26-27] that uses static-CMOS PLAs [56-57]. In order to provide truly power-friendly CPLD architectures, we would need to develop a completely new architecture based on static-CMOS gates rather than pseudo-nMOS gates. This would require a very large design effort, including new layout units, layout generation code, area models, delay models, and power models. The effort may be worthwhile, however, as the increasing feasibility of SoC devices is likely to cause an increase in their deployment in low power devices such as PDAs and cell phones in upcoming years.

One design decision that had a negative impact was our choice to use an executable version of PLAmapping in our tool flow for creating domain-specific CPLDs. The first problem was that PLAmapping would sometimes fail to provide us with results. This not only led to slightly suboptimal results for the data points where PLAmapping failed, but it also required a significant amount of user intervention in order to create the missing data.

Another problem with using an executable of PLAmapping was that it allowed us very little control of the algorithm itself. We were not able to obtain fine-grained control of delay and area tradeoffs for the algorithms, or make any modifications to the mapping algorithms that might be useful, such as providing heuristics that give us faster results at a slight penalty to mapping quality. This would have been useful for the circuits that typically required an hour or more for a single run of PLAmapping.

The main reason that we are creating domain-specific architectures is that we want to leverage the similarities that exist between circuits in a domain. While PLAmapping is the best academic technology-mapping algorithm available for our use, it is not particularly well suited to creating domain-specific architectures. PLAmapping's algorithm simply tries to obtain a minimum-depth mapping, and then uses heuristics to pack PLAs into each other to minimize the area of a mapping. The domain-specific gains that we achieved were likely due to fairly coarse-grained similarities that existed between circuits in a domain, such as the primary input count, the primary output count, and the number of levels of logic existing between registers. PLAmapping is unable to look closely at the structure of different circuits in order to find similarities that it can exploit, which is a distinct drawback of the algorithm.

If any future work is performed on the creation of domain-specific CPLD architectures, I believe that a new tech-mapping algorithm should be developed for use in the system. This would allow us to have complete control over the algorithms in the technology-mapper, as well as allowing us to implement new algorithms that are able to identify and utilize similarities between different circuits.

The proposed tech-mapper would be able to look at multiple circuits and find regions of similarities that can be mapped to similar PLA structures. This would allow us to better leverage the similarities that exist between circuits in a domain. I also propose that the tech-mapping algorithm should be allowed to use PLAs of varying size within the CPLD architecture, as different similarities that are extracted from the circuits are likely to want to map to differently sized PLAs. This might not provide area gains due to the fact that the CPLD's bounding box is determined by the largest PLAs in the architecture, but it would provide performance gains due to reduced capacitance in the architectures. When put into a tool flow, the proposed tech-mapper would first perform these domain-specific optimizations in order to extract similarities from the circuits, and would then use a PLAmapping-like algorithm in order to map the remaining logic to PLAs. I believe that these modifications would result in CPLD architectures that more accurately represent the domains that they support.

In terms of the bigger picture, we now need to determine the next direction in which to take Totem. One aspect of CPLD architectures is that their interconnect structures limit them to implementing relatively small circuits. Totem-CPLD can therefore be seen as a method of creating relatively low capacity domain-specific reconfigurable architectures, using relatively fine-grained functional units. Earlier work in Totem leveraged RaPiD, a 1-D architecture targeted at the signal-processing domain. Totem-RaPiD can provide us with a method for creating reconfigurable architectures for implementing coarse-grained datapaths, but it is inefficient at supporting any fine-grained circuits. A gap still exists in the Totem family, as there is currently no method of creating large and effective reconfigurable fabrics that tailor to small-to-medium-grained applications.

I believe that the next forays into Totem should address this issue. Since 1-D architectures seem to be rather limited in their scope (RaPiD) or scalability (CPLDs), this would suggest the creation of a 2-D reconfigurable architecture, most likely of an island-style nature. The trend in commercial devices is clearly toward these 2-D reconfigurable architectures, especially when considering devices with reasonably large capacity.

The difficulties involved with creating a domain-specific 2-D architecture might outweigh the foreseeable gains, however. Tile-based reconfigurable architectures are already available for SoC designers, and the companies that provide them have put a large amount of time and money into creating efficient (though general) architectures and high-quality layouts. While domain-specific 2-D architectures would hypothetically provide performance gains over these more general fabrics, it is unclear whether the performance gained by tailoring to a domain would overcome the performance lost in terms of architecture and layout quality.

Additionally, domain-specific optimizations will be more difficult to make in 2-D architectures, because there are more architectural constraints. As an example, consider the basic island style FPGA diagram shown on the left half of Figure 98. The architecture has horizontal and vertical patterns that are very regular, such that changing the size of any single item in the architecture will have an effect on both entire row and

column in which the item resides. On the right side of Figure 98 we have made just one of the logic blocks larger, and it has left us with a large amount of unused space in the horizontal and vertical directions (shaded).

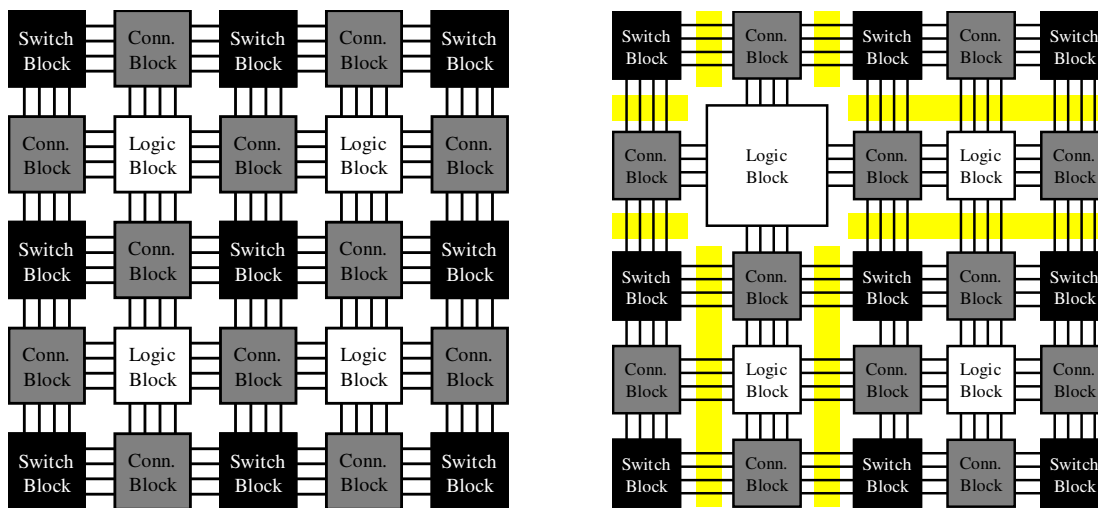


Figure 98. On the left, a basic diagram of a 2-D island style FPGA. On the right, the effects of changing a single piece of the architecture. Note the extra unutilized space in the column and row that the larger logic block resides in (the shaded regions in the diagram)

Clearly, we are limited in the types of optimizations that could be made in such an architecture. In order to retain compact layouts, the logic blocks would all need to have footprints of similar size. Another restriction is that routing channels would have to be of uniform size across the entire width or height of the architecture in order to retain compact implementations. These layout constraints will make it difficult for the architecture generation process to make many of the optimizations that a designer might desire.

Another consideration is that, since we are now competing closely with tile-based reconfigurable IP solutions, we will require very high-quality layouts for these 2-D domain-specific architectures. This basically rules out the use of standard cell methods, as the performance penalties imposed by these methods would almost certainly be too high. In order to leverage the high regularity of the 2-D structure, I believe that an additive layout generation process must be developed which tiles efficient layouts of the logic, switch, and connection blocks. This will require either the creation of very

efficient circuit generators, or the pre-designing of highly optimized cells that can be tiled into a full architecture. In either method, a large amount of design effort will need to go into the generation of 2-D layouts.

Despite being highly constrained, however, interesting things can definitely be done within the framework of a domain-specific 2-D architecture. Clustering in the logic blocks would allow different logic implementation schemes, including different LUT sizes, small PLA blocks, small ALUs, and any other small units that are deemed appropriate. Routing channels can contain different numbers of tracks, and switch and connection blocks can contain different connectivity.

The big question, though, will be whether a domain-specific 2-D architecture would be able to compete with and beat current tile-based reconfigurable fabrics. Because of the large amount of effort that Totem-2D will require, I think that a detailed analysis should first be undertaken in order to determine the feasibility of such an endeavor. If an analysis shows that Totem-2D can provide gains over other solutions that currently exist, then I believe that it should be explored.

If Totem-2D turns out to be infeasible, however, another future option for Totem is to revisit the work that was done with RaPiD-AES. RaPiD-AES was a RaPiD-style architecture that introduced new logic units that were optimized to private-key encryption [12]. Work on RaPiD-AES was halted when the domain-specific results provided by RaPiD-AES were found to be significantly worse than Verilog implementations on a standard Xilinx FPGA.

In a recent Totem paper [25], RaPiD-AES is hypothesized to have failed largely because it didn't follow the strategy of "flexible-first". The idea is that reconfigurable architectures should start with a flexible fabric and add fixed functionality only as it proves beneficial. RaPiD-AES didn't follow this methodology, as it provided only coarse-grained functional units, which were often underutilized. In dealing with the creating of domain-specific PLAs and PALs, this dissertation has basically introduced PLA and PAL generators that could be used to create flexible units for RaPiD-AES. By

introducing these flexible units, we could take another look at the practicality of RaPiD-AES architectures, as well as investigating the merit of adding “flexible-first”.

References

- [1] C. Ebeling, D.C. Cronquist, and P. Franklin, “RaPiD – Reconfigurable Pipelined Datapath.”, *Lecture Notes in Computer Science 1142—Field-Programmable Logic: Smart Applications, New Paradigms and Compilers*, R.W. Hartenstein, M. Glesner, Eds. Springer-Verlag, Berlin, Germany, pp. 126-135, 1996.
- [2] D. C. Cronquist, P. Franklin, C. Fisher, M. Figueroa, C. Ebeling, “Architecture Design of Reconfigurable Pipelined Datapaths”, *Twentieth Anniversary Conference on Advanced Research in VLSI*, 1999.
- [3] A. Abnous, J. Rabaey, “Ultra-low-power domain-specific multimedia processors.”, *VLSI Signal Processing, IX*, IEEE Signal Processing Society, pp. 461-470, 1996.
- [4] S. Goldstein, H. Schmidt, M. Budiu, S. Cadambi, M. Moe, R. Taylor, “PipeRench: A Reconfigurable Architecture and Compiler”, *IEEE Computer*, 33(4):70-77, April 2000.
- [5] Xilinx, Inc, “Xilinx: Products & Services: Silicon Solutions: Virtex-4 Overview”, <<http://www.xilinx.com/products/virtex4/overview.htm>> (2005).
- [6] K. Compton, S. Hauck, “Totem: Custom Reconfigurable Array Generation”, *IEEE Symposium on FPGAs for Custom Computing Machines*, 2001.
- [7] K. Compton, A. Sharma, S. Phillips, S. Hauck, “Flexible Routing Architecture Generation for Domain-Specific Reconfigurable Subsystems”, *International Conference on Field Programmable Logic and Applications*, pp. 59-68, 2002.
- [8] K. Compton, *Architecture Generation of Customized Reconfigurable Hardware*, Ph.D. Thesis, Northwestern University, Dept. of ECE, 2003.
- [9] K. Compton, S. Hauck, “Track Placement: Orchestrating Routing Structures to Maximize Routability”, *International Conference on Field Programmable Logic and Applications*, 2003.
- [10] K. Compton, S. Hauck, “Flexibility Measurement of Domain-Specific Reconfigurable Hardware”, *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, pp. 155-161, 2004.

- [11] K. Compton, S. Hauck, "Automatic Design of Configurable ASICs", submitted to *IEEE Transactions on VLSI Systems*.
- [12] K. Eguro, *RaPiD-AES: Developing an Encryption-Specific FPGA Architecture*, Master's Thesis, University of Washington, Dept. of EE, 2002.
- [13] K. Eguro, S. Hauck, "Issues and Approaches to Coarse-Grain Reconfigurable Architecture Development", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 111-120, 2003.
- [14] K. Eguro, S. Hauck, "Resource Allocation for Coarse Grain FPGA Development", to appear in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, October, 2005.
- [15] M. Holland, S. Hauck, "Automatic Creation of Reconfigurable PALs/PLAs for SoC", *International Conference on Field Programmable Logic and Applications*, pp. 536-545, 2004.
- [16] M. Holland, S. Hauck, "Automatic Creation of Domain-Specific Reconfigurable CPLDs for SoC", *International Conference on Field Programmable Logic and Applications*, 2005.
- [17] S. Phillips, *Automatic Layout of Domain-Specific Reconfigurable Subsystems for System-on-a-Chip*, M.S. Thesis, Northwestern University, Dept. of ECE, July 2001.
- [18] S. Phillips, S. Hauck, "Automatic Layout of Domain-Specific Reconfigurable Subsystems for System-on-a-Chip", *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, pp. 165-173, 2002.
- [19] S. Phillips, *Automating Layout of Reconfigurable Subsystems for Systems-on-a-Chip*, Ph.D. Thesis, University of Washington, Dept. of EE, 2004.
- [20] S. Phillips, A. Sharma, S. Hauck, "Automating the Layout of Reconfigurable Subsystems Via Template Reduction", *International Conference on Field Programmable Logic and Applications*, pp. 857-861, 2004.
- [21] S. Phillips, S. Hauck, "Automating the Layout of Reconfigurable Subsystems Using Circuit Generators", *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005.
- [22] A. Sharma, *Development of a Place and Route Tool for the RaPiD Architecture*, M.S. Thesis, University of Washington, 2001.
- [23] A. Sharma, *Place and Route Techniques for FPGA Architecture Advancement*, Ph.D. Thesis, University of Washington, Dept. of EE, 2005.

- [24] A. Sharma, C. Ebeling, S. Hauck, "PipeRoute: A Pipelining-Aware Router for FPGAs", *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, pp. 68-77, 2003.
- [25] S. Hauck, K. Compton, K. Eguro, M. Holland, S. Phillips, A. Sharma, "Totem: Domain-Specific Reconfigurable Logic", submitted to *IEEE Transactions on VLSI Systems*.
- [26] Xilinx, Inc., *CoolRunner XPLA3 CPLD: Preliminary Product Specification*, January 6, 2003.
- [27] Xilinx, Inc., *CoolRunner-II CPLD Family: Advance Product Specification*, March 12, 2003.
- [28] Xilinx, Inc., *XC9500XV Family High-Performance CPLD: Preliminary Product Specification*, June 24, 2002.
- [29] Altera Corporation, *Configuration Elements: Data Sheet*, January 1988.
- [30] D. Chen, J. Cong, M. Ercegovac, Z. Huang, "Performance-Driven Mapping for CPLD Architectures", *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, 2001.
- [31] J.H. Anderson, S.D. Brown, "Technology Mapping for Large Complex PLDs", *Proceedings of the 35th ACM/IEEE Design Automation Conference*, pp. 698-703, 1998.
- [32] N. Kafafi, K. Bozman, S. Wilton, "Architectures and Algorithms for Synthesizable Embedded Programmable Logic Cores", *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, 2003.
- [33] A. Yan, S. Wilton, "Product-Term Based Synthesizable Embedded Programmable Logic Cores", *IEEE International Conference on Field-Programmable Technology*, pp. 162-169, 2003.
- [34] F. Mo, R. Brayton, "River PLAs: A Regular Circuit Structure", *Proceedings of the 39th ACM/IEEE Design Automation Conference*, 2002.
- [35] Y. Han, L. McMurchie, C. Sechen, "A High Performance Programmable Logic Core for SoC Applications", *13th ACM International Symposium on Field-Programmable Gate Arrays*, 2005.
- [36] Actel Corporation, "Varicore", <<http://www.actel.com/varicore/products/index.html>> (2005).

- [37] Elixent, “Reconfigurable Compute/Processing Array – the D-Fabrix”, <<http://www.elixent.com/products/array.htm>> (2005).
- [38] Elixent, “matsushita”, <http://www.elixent.com/press_area/pressreleases/matsushita.htm> (April 6, 2005).
- [39] Xilinx, Inc., *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet: Product Specification*, June 20, 2005.
- [40] Xilinx Inc., “Xilinx: Products & Services: Silicon Solutions: Virtex-4 FPGAs: DSP Applications”, <<http://www.xilinx.com/products/virtex4/overview/dsp.htm>> (2005).
- [41] S. Khawam, T. Arslan, F. Westall, “Embedded Reconfigurable Array Targeting Motion Estimation Applications”, 2003.
- [42] M. Abramovici, C. Stroud, M. Emmert, “Using Embedded FPGAs for SoC Yield Improvement”, *Proceedings of the 39th ACM/IEEE Design Automation Conference*, 2002.
- [43] B. Quinton, S. Wilton, “Post-Silicon Debug Using Programmable Logic Cores”, *IEEE International Conference on Field-Programmable Technology*, 2005.
- [44] R. K. Brayton, G. D. Hachtel, C. T. McMullen, A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Boston, Kluwer Academic Publishers, 1984.
- [45] “1993 LGSynth Benchmarks”, <http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.cbl.ncsu.edu/CBL_Docs/lgs93.html> (March 25, 1997).
- [46] J. P. Uyemura, *CMOS Logic Circuit Design*, Boston, Kluwer Academic Publishers, 1999.
- [47] OpenCores.org, “OPENCORES.ORG”, <<http://www.opencores.org/>> (2004).
- [48] M. Leaser, “Variable Precision Floating Point Modules”, <<http://www.ece.neu.edu/groups/rpl/projects/floatingpoint/>> (May 20, 2004).
- [49] National Institute of Standards and Technology, *FIPS PUB 197, Advanced Encryption Standard (AES)*, November 2001.
- [50] Asratian, Denley, Häggkvist, *Bipartite Graphs and Their Applications*, UK, Cambridge University Press, 1998.

- [51] V. Betz, J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research", *International Workshop on Field Programmable Logic and Applications*, 1997.
- [52] J. Kouloheris, A. El Gamal, "FPGA Performance vs. Cell Granularity", *Proceedings of the Custom Integrated Circuits Conference*, 1991.
- [53] G. Lemieux and D. Lewis, *Design of Interconnection Networks for Programmable Logic*, Boston, Kluwer Academic Publishers, 2004.
- [54] A. El Gamal, L.A. Heinachandra, I. Shperling, V. K. Wei, "Using simulated annealing to design good codes.", *IEEE Transactions on Information Theory*, 33(1):116:123, January 1987.
- [55] L. McMurchie, C. Ebeling, "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs", *Proceedings of the 1995 ACM Third International Symposium on Field-Programmable Gate Arrays*, pp. 111-117, February 1995.
- [56] Xilinx, Inc, "Technology Update - CoolRunner", <http://www.xilinx.com/xcell/xl36/xl36_10.pdf>.
- [57] Xilinx, Inc, "Fast Zero Power (FZP™) Technology", <<http://www.xilinx.com/products/coolpld/wp119.pdf>>, 2000.

Appendix A: Kuhn/Munkres Algorithm

This Appendix details an algorithm by Kuhn and Munkres that can be used to map the product terms of two circuits onto PLA and PAL arrays such that they require the minimal number of programmable connections. This is done by rephrasing the problem as a problem on a bipartite graph with weighted edges, and finding the best perfect matching. The algorithm for finding the best perfect matching is taken from [50].

The process begins with the product terms of each circuit. If one of the circuits contains fewer product terms than the other circuit, then it must be filled with empty product terms in order to contain the same number of product terms as the second circuit. This is shown in Figure 99.

<i>circuit1</i>	<i>circuit2</i>	
010010110	101000000	
101010010	110010001	
010110000	011101011	
100100011	000000000	} added terms
011011101	000000000	

Figure 99. If one of the circuits has fewer product terms, then it is filled with empty product terms in order to contain the same number as the other circuit. In this figure the product terms use their vector representations, as introduced in Chapter 8

Next, the value of all possible product term matchings must be determined. The value of a product term matching is the number of array locations that do not need programmable connections when the two product terms are mapped to the same location. It is formulated in this fashion so that we have a maximization problem rather than a minimization problem.

<i>circuit1</i>	<i>circuit2</i>	Weight Table																																				
a:010010110	v:101000000	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th style="width: 20px;"></th> <th style="width: 20px;">v</th> <th style="width: 20px;">w</th> <th style="width: 20px;">x</th> <th style="width: 20px;">y</th> <th style="width: 20px;">z</th> </tr> </thead> <tbody> <tr> <th style="width: 20px;">a</th> <td>3</td> <td>3</td> <td>1</td> <td>5</td> <td>5</td> </tr> <tr> <th style="width: 20px;">b</th> <td>5</td> <td>3</td> <td>1</td> <td>5</td> <td>5</td> </tr> <tr> <th style="width: 20px;">c</th> <td>4</td> <td>4</td> <td>2</td> <td>6</td> <td>6</td> </tr> <tr> <th style="width: 20px;">d</th> <td>4</td> <td>3</td> <td>2</td> <td>5</td> <td>5</td> </tr> <tr> <th style="width: 20px;">e</th> <td>2</td> <td>3</td> <td>1</td> <td>3</td> <td>3</td> </tr> </tbody> </table>		v	w	x	y	z	a	3	3	1	5	5	b	5	3	1	5	5	c	4	4	2	6	6	d	4	3	2	5	5	e	2	3	1	3	3
	v		w	x	y	z																																
a	3		3	1	5	5																																
b	5		3	1	5	5																																
c	4		4	2	6	6																																
d	4	3	2	5	5																																	
e	2	3	1	3	3																																	
b:101010010	w:110010001																																					
c:010110000	x:011101011																																					
d:100100011	y:000000000																																					
e:011011101	z:000000000																																					

Figure 100. The matching values are calculated for each possible product term matching between two circuits. The Weight Table displays the values for the matchings

The problem is now recast as a problem on a bipartite graph, G . The product terms from *circuit1* become vertices in the left partition (V_1) of G , and the product terms from *circuit2* become vertices in the right partition (V_2) of G . For each vertex u in V_1 , an edge is created to each vertex v in V_2 with a weight $w(uv)$ equaling the value calculated for the corresponding product-term matching. An example of this is shown in Figure 101. The problem is now to find the perfect matching between V_1 and V_2 whose edges sum to a maximum value.

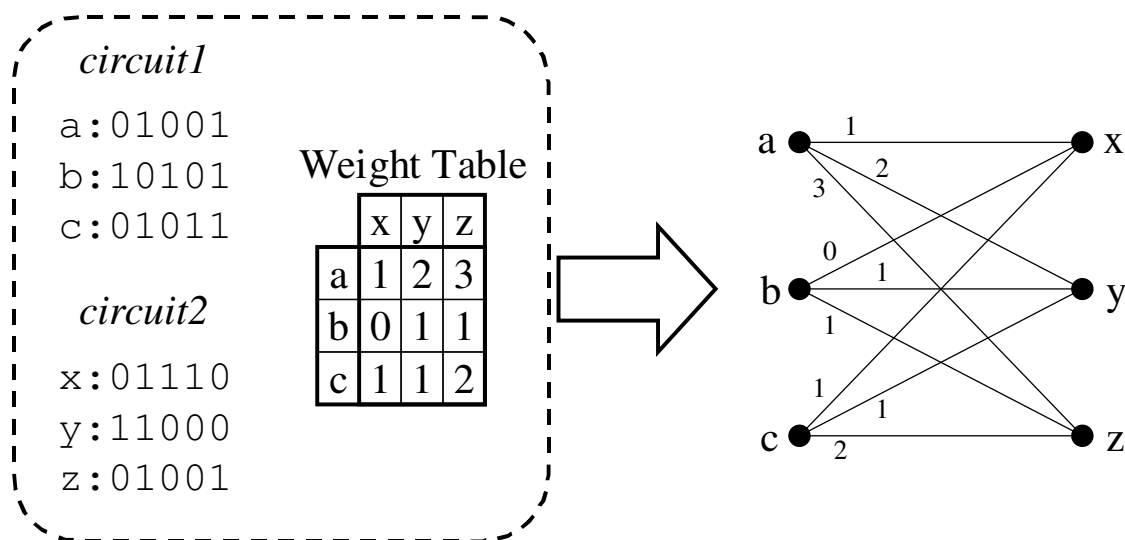


Figure 101. Casting the problem into the problem of finding the maximal perfect matching on a bipartite graph

We define a function f which obeys $f(u) + f(v) \geq w(uv)$ for each $u \in V_1, v \in V_2$, and instantiate the function as follows:

$$f_0(x) = \begin{cases} \max\{w(xy) : y \in N(x)\} & \text{if } x \in V_1 \\ 0 & \text{if } x \in V_2 \end{cases}$$

where $N(x)$ is the neighborhood of x . Basically, for each vertex in V_1 , f is set to the value of the maximum edge connected to the vertex. For each vertex in V_2 , f is set to 0. An example of this is shown in Figure 102.

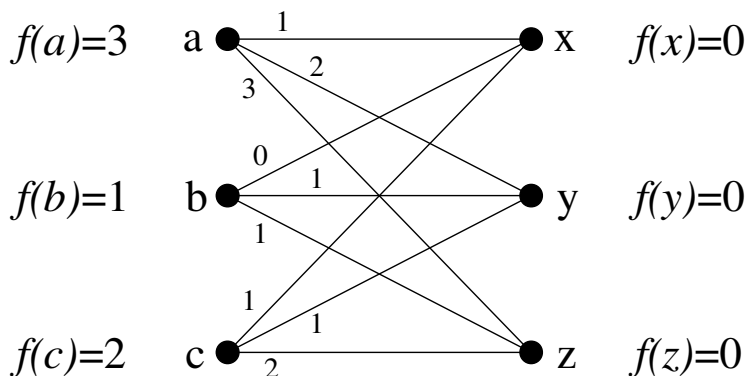


Figure 102. Example of initial values for the function f

We next create the spanning subgraph of G , called G_f , which contains the subset of edges uv for which $f(u)+f(v)=w(uv)$. G_f therefore contains all the vertices from G , but only some of the edges. An example of this subgraph G_f is shown in Figure 103.

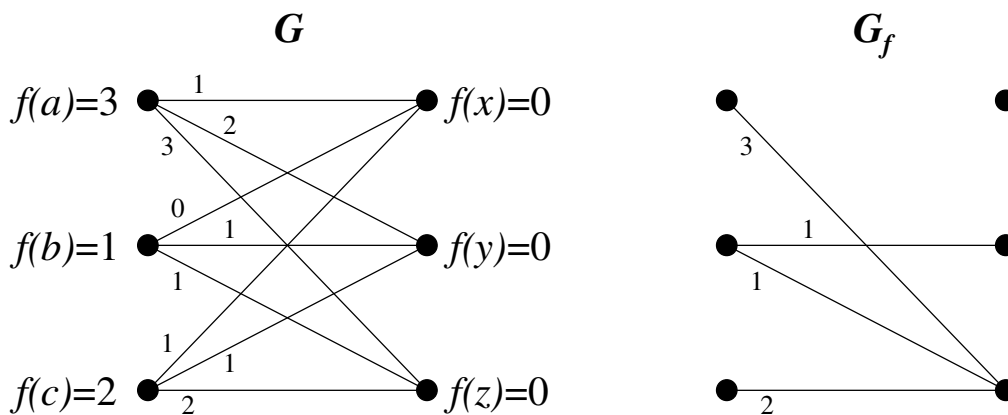


Figure 103. On the left, the graph G with the values of f for each vertex. On the right, the spanning subgraph G_f that is created from G

The problem has now been set up, and the Kuhn/Munkres algorithm can be used find the optimal assignment on the bipartite graph, which will give us the optimal product term assignment. The pseudocode for the Kuhn/Munkres algorithm is shown in Figure 104.

```

KuhnMunkresAlgorithm()
{
    • Begin with the feasible labeling  $f = f_0$ , the equality graph  $G_f$ , and
      a maximum matching of  $G_f$ , called  $M_0$ , which is initially empty. Let
       $i = 0$ .
    • While  $M_i$  is not a perfect matching of  $G_f$ 
      ◦ Let  $u$  be a free vertex in  $V_1$ . Let  $S = \{u\}$  and  $T = \emptyset$ .
      (1) ◦ If  $N_{G_f}(S) = T$  then
          ▪ Compute  $d_f = \min_{x \in S, y \notin T} \{f(x) + f(y) - w(xy)\}$ 
          ▪ Update the function  $f$  to
              
$$f(x) = \begin{cases} f(x) - d_f & \text{if } x \in S, \\ f(x) + d_f & \text{if } x \in T, \\ f(x) & \text{if otherwise,} \end{cases}$$

              and calculate the new  $G_f$ .
          ◦ Select a vertex  $y \in N_{G_f}(S) \setminus T$ .
          ◦ If  $y$  is saturated then add  $y$  to  $T$ , and the neighbor of  $y$  in
               $M_i$  to  $S$ . Goto (1).
          ◦ Else there is an augmenting path  $P$  in  $G_f[S \cup T]$  joining  $u$  to
               $y$ . Let  $M_{i+1} = M_i \Delta E(P)$  and  $i = i + 1$ .
    • End while
    • Output  $M_i$ , an optimal matching.
}

```

Figure 104. The Kuhn/Munkres algorithm for finding the optimal matching in a bipartite graph [50]

The algorithm runs by searching for augmenting paths relative to the current matching. When new augmenting paths are not found, it alters the function f in order to modify the graph G_f , which allows new augmenting paths to be found. For circuits with

n product terms, there can be at most n matchings in M_i . For each matching, the set T can be increased up to n times, which causes an update to f which can take $O(n^2)$ steps. The overall runtime of the algorithm is thus $O(n^4)$.

Appendix B: Layout Units

This Appendix presents the major layout units that are used to build our CPLD architectures, including their schematic representations. For brevity's sake, only layout units that utilize silicon are shown: any layout units that are comprised purely of routing resources (metal layers) are omitted from this appendix. Additionally, many layout units come in both a standard and upside-down form: this appendix will only display their standard forms.

Figure 105 displays the very corner of a CPLD that our tool might create. Included in this diagram is a very small (2-4-2) PLA, along with a row of programmable switches that would be part of the crossbar that switches signals into the PLA. The figure also depicts the individual layout units that are tiled in order to create the CPLD structure. Each of the individual units displayed in Figure 105 is shown in this Appendix, along with a schematic representation of the unit.

Signals are switched from the CPLD interconnect to a PLA through a programmable switch, shown in Figure 106. The programmable switch consists of a transistor whose gate is controlled by an SRAM bit. The output signal of this programmable switch feeds an inverter and buffer, shown in Figure 107. This unit provides the true and negated form of a signal to the AND-plane in the PLA.

Each product term in the PLA AND-plane is a pseudo-nMOS gate. The pulldown section of the pseudo-nMOS gate is created by the AND-plane cells, which consist of pulldown transistors that are either connected or disconnected from ground according to SRAM bits (Figure 108). The pullup transistor for the pseudo-nMOS gate is shown in Figure 109. This signal is then restored/amplified by a buffer, shown in Figure 110.

The outputs of the PLA are created in the OR-plane, and they are also formed using pseudo-nMOS gates. The OR-plane cells (Figure 111) contain the pulldown transistors

for the pseudo-nMOS gates, and the pullup transistor is shown in Figure 112. This signal is then inverted and amplified using the cell in Figure 113: inversion is necessary due to our NOR-NOR PLA implementation.

The output of the PLA then feeds a D-Flip-Flop (Figure 114), which is used to provide registering for the signal. A 2-to-1 multiplexor (Figure 115) then chooses between the registered and unregistered PLA output signal, as determined by an SRAM bit, shown in Figure 116. This signal is then connected to a specific wire in the CPLD interconnect, as required by our complete network implementation.

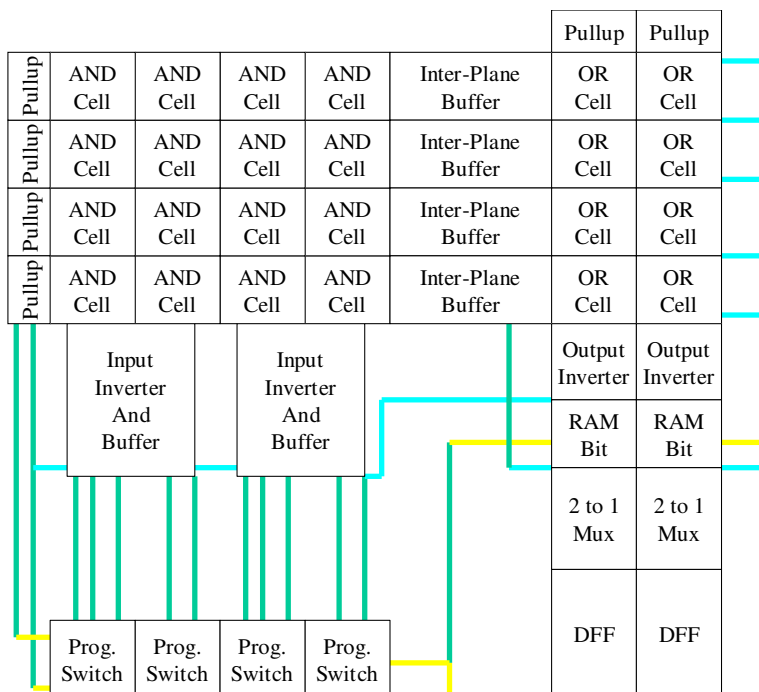
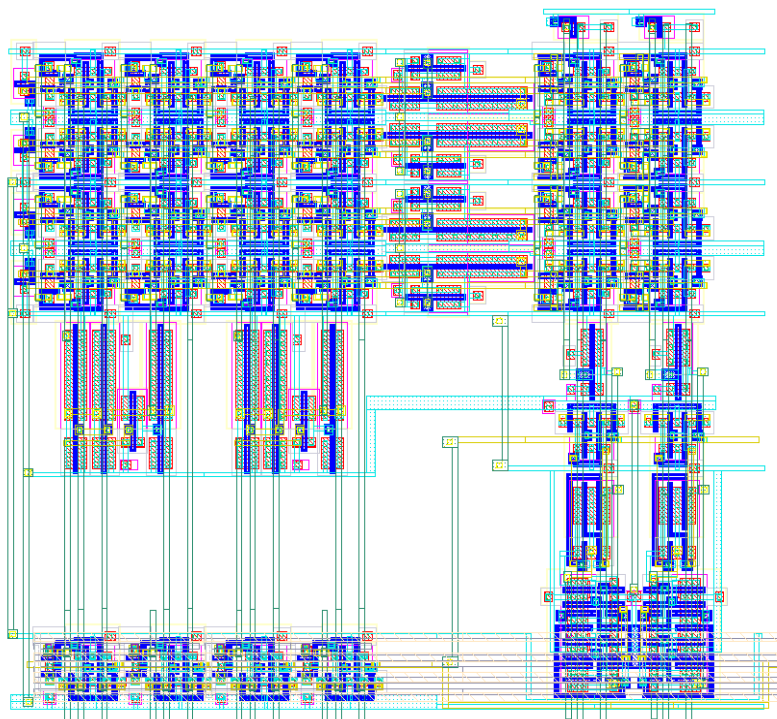


Figure 105. Top, layout of a PLA and programmable switches (part of the crossbar) from a CPLD that our tool creates. Bottom, the main cells used in creating this part of the layout

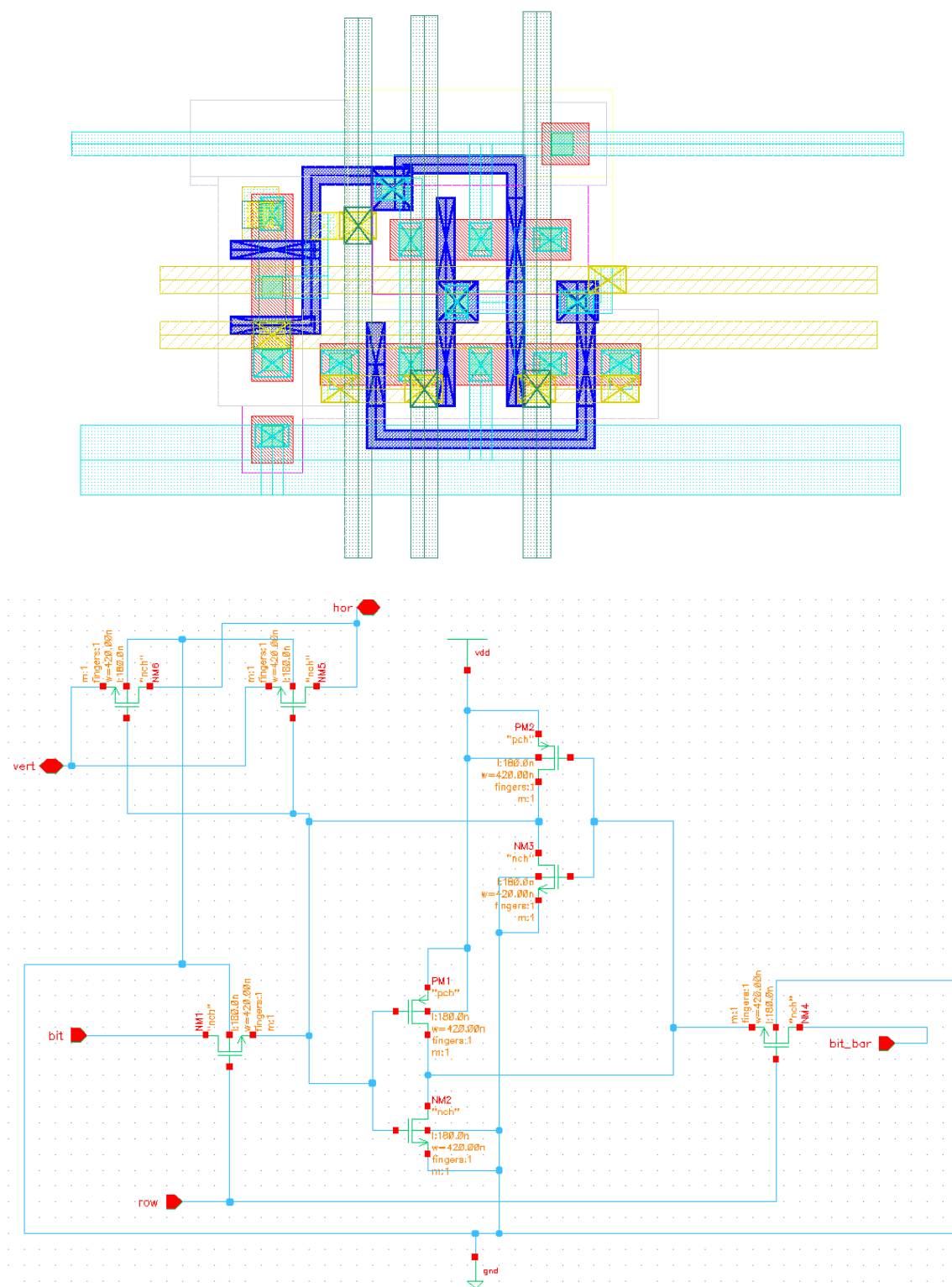


Figure 106. Programmable switch from the CPLD crossbar

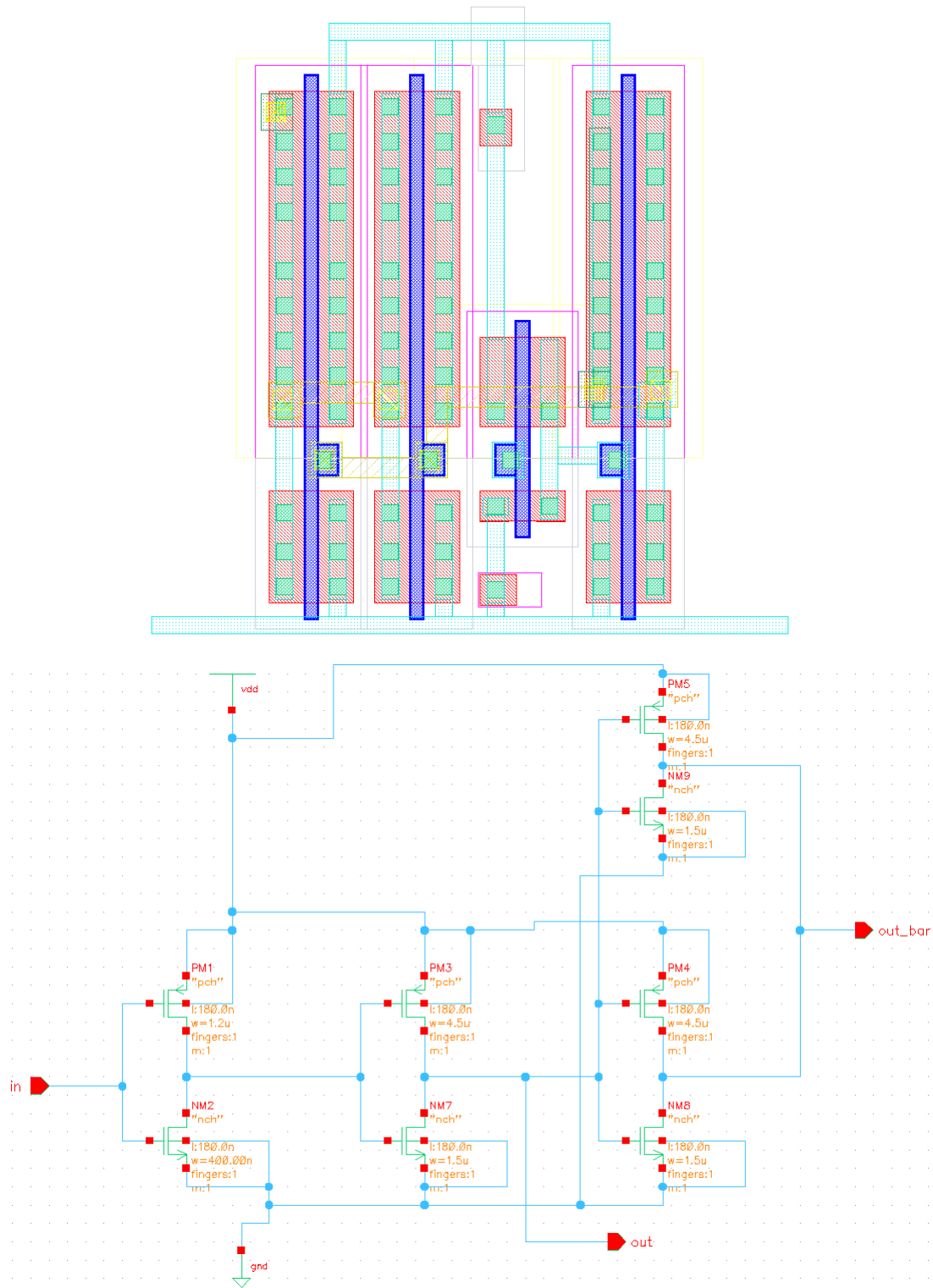


Figure 107. Inverter and buffer that feed the PLA's AND-plane

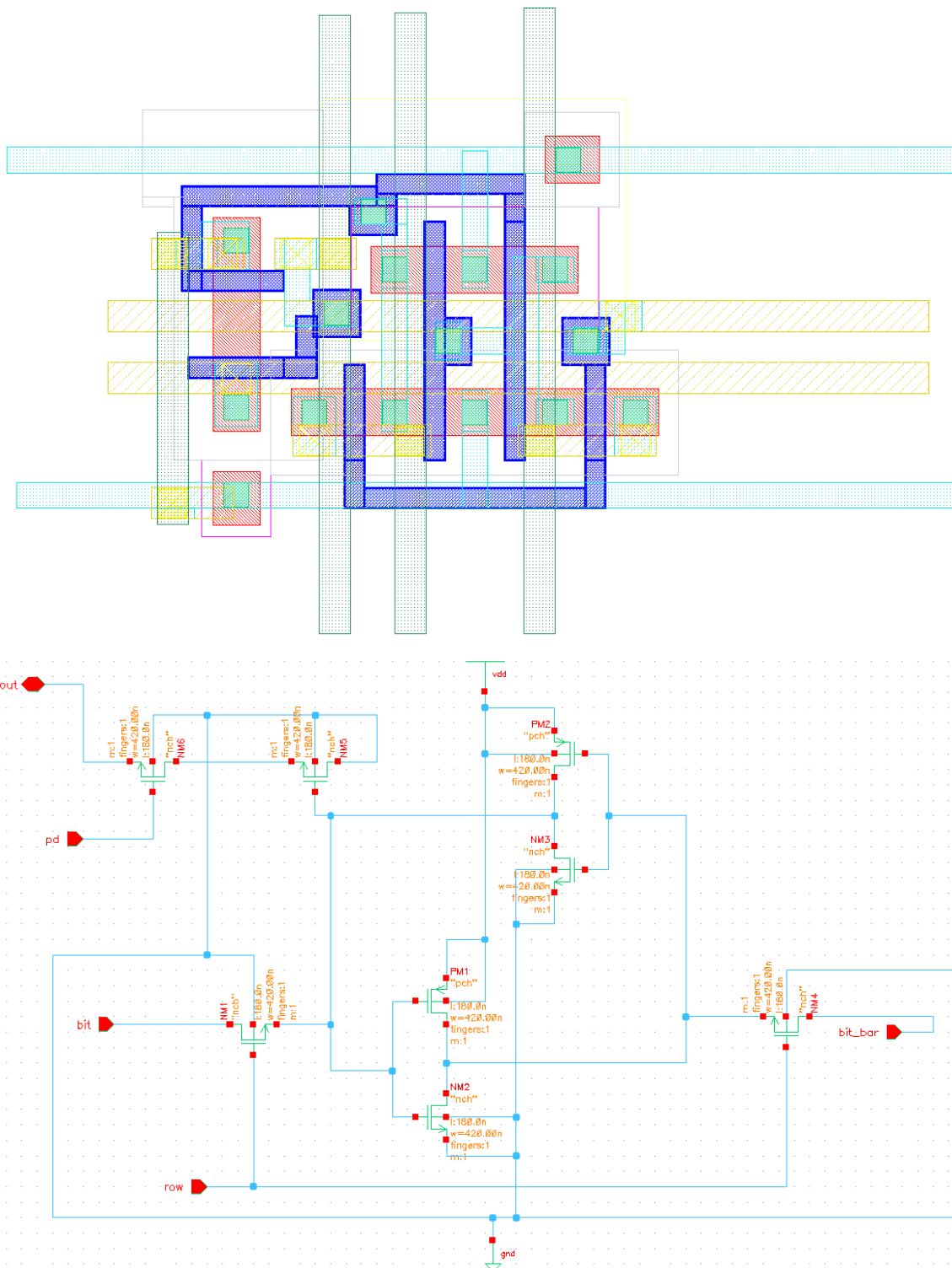


Figure 108. AND-plane connection, a pulldown transistor in series with an SRAM controlled transistor

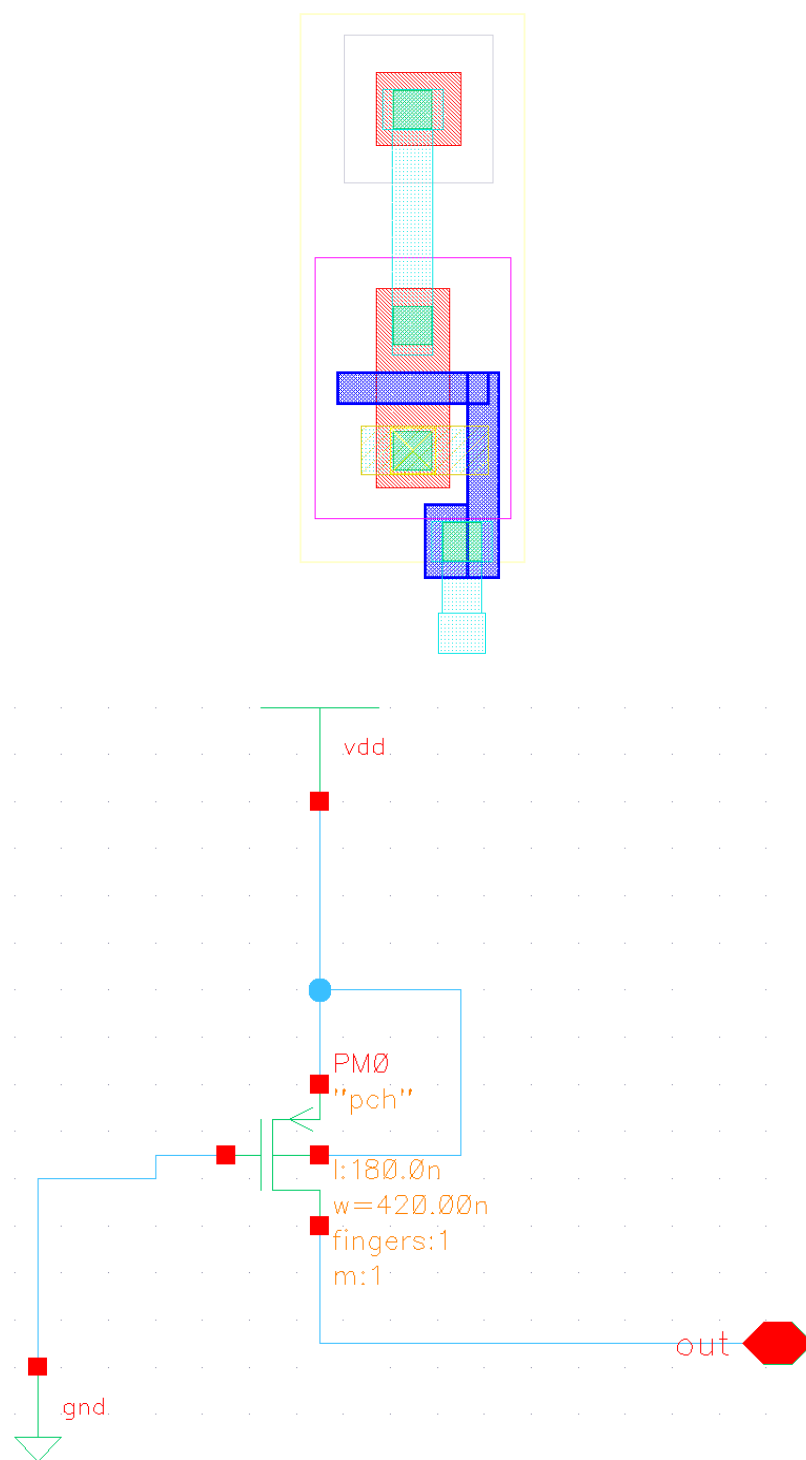


Figure 109. Pullup transistor for the AND-plane

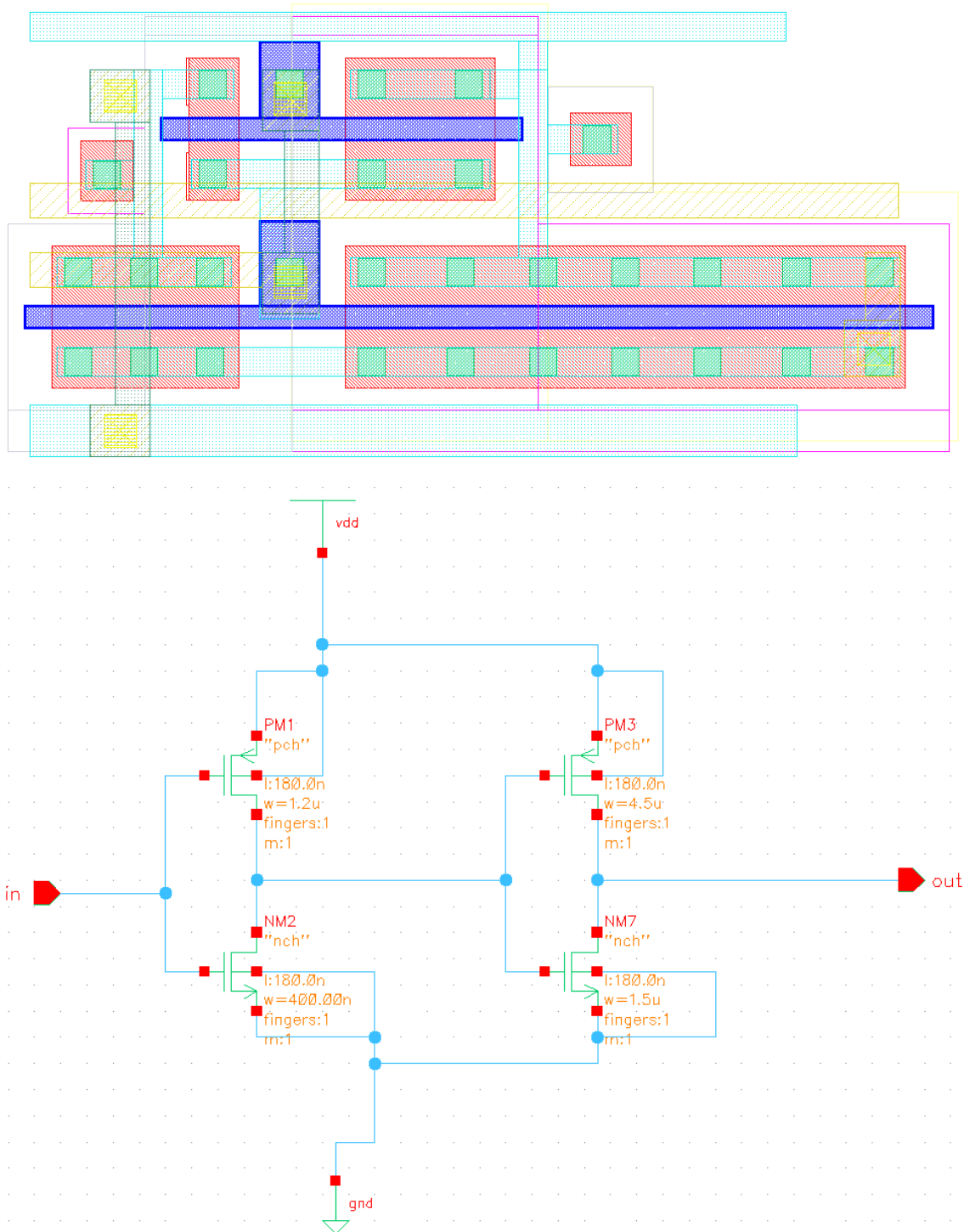


Figure 110. Buffer between the AND-plane and OR-plane, used for signal restoration

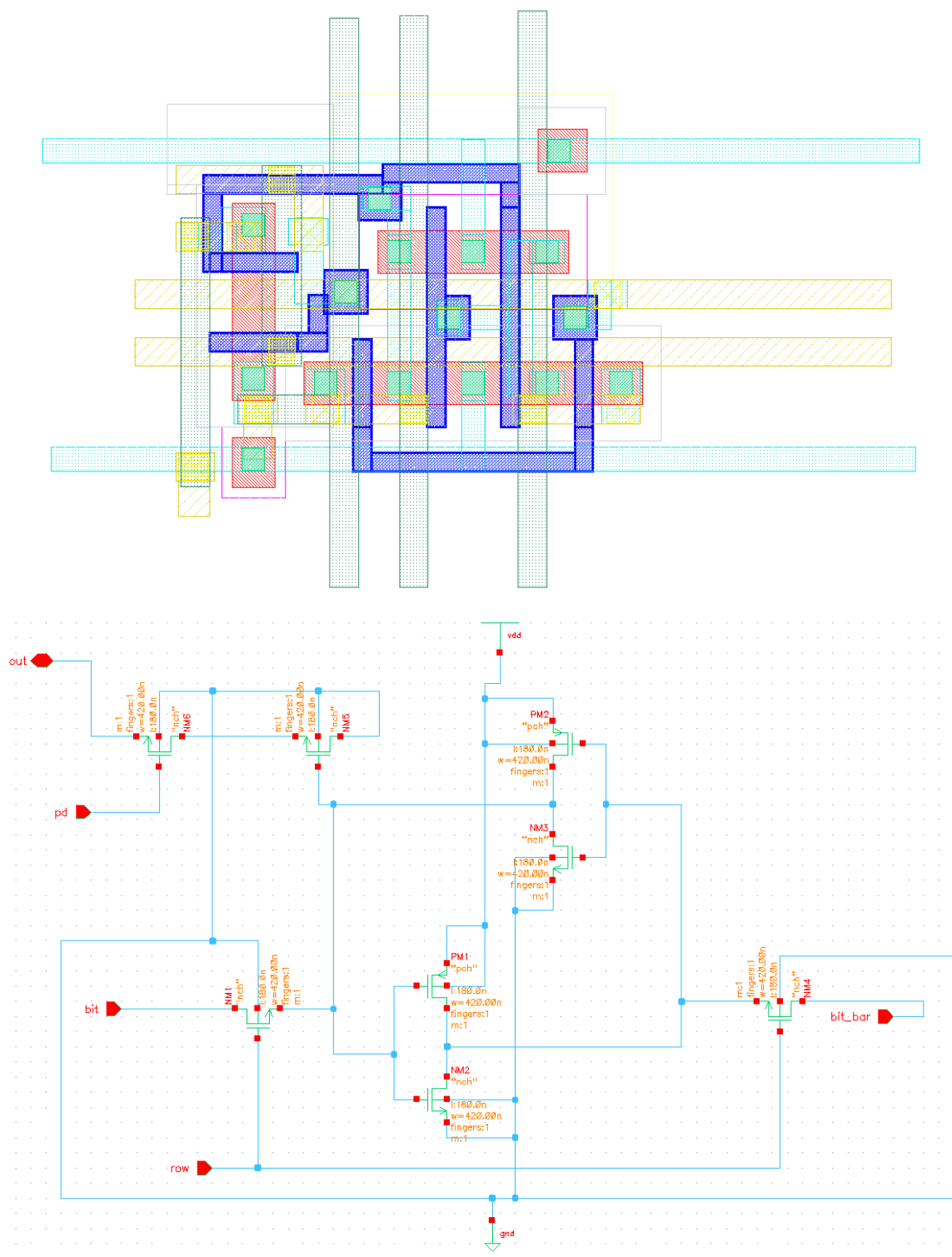


Figure 111. OR-plane connection, a pulldown transistor in series with an SRAM controlled transistor

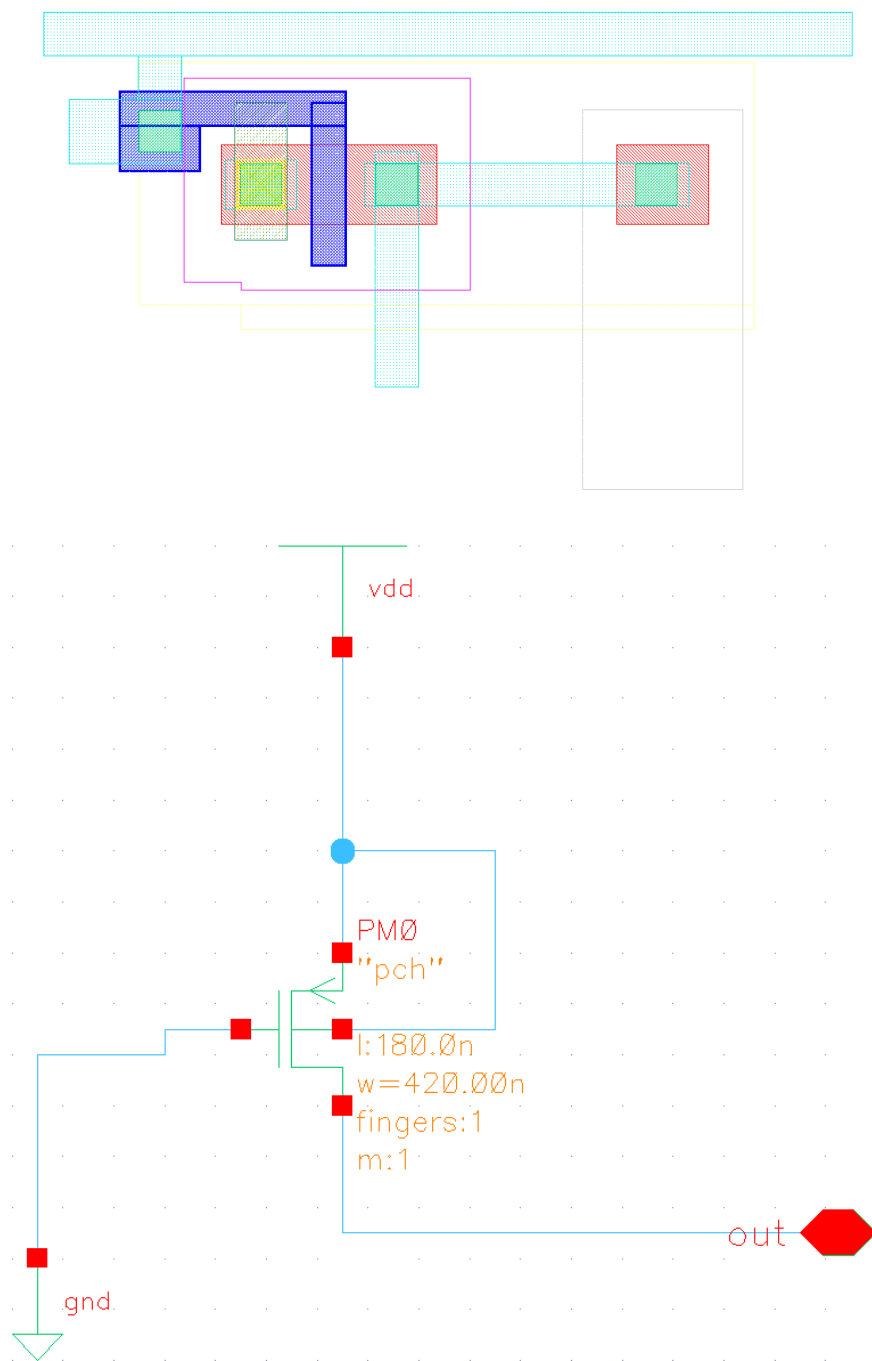


Figure 112. Pullup transistor for the OR-plane

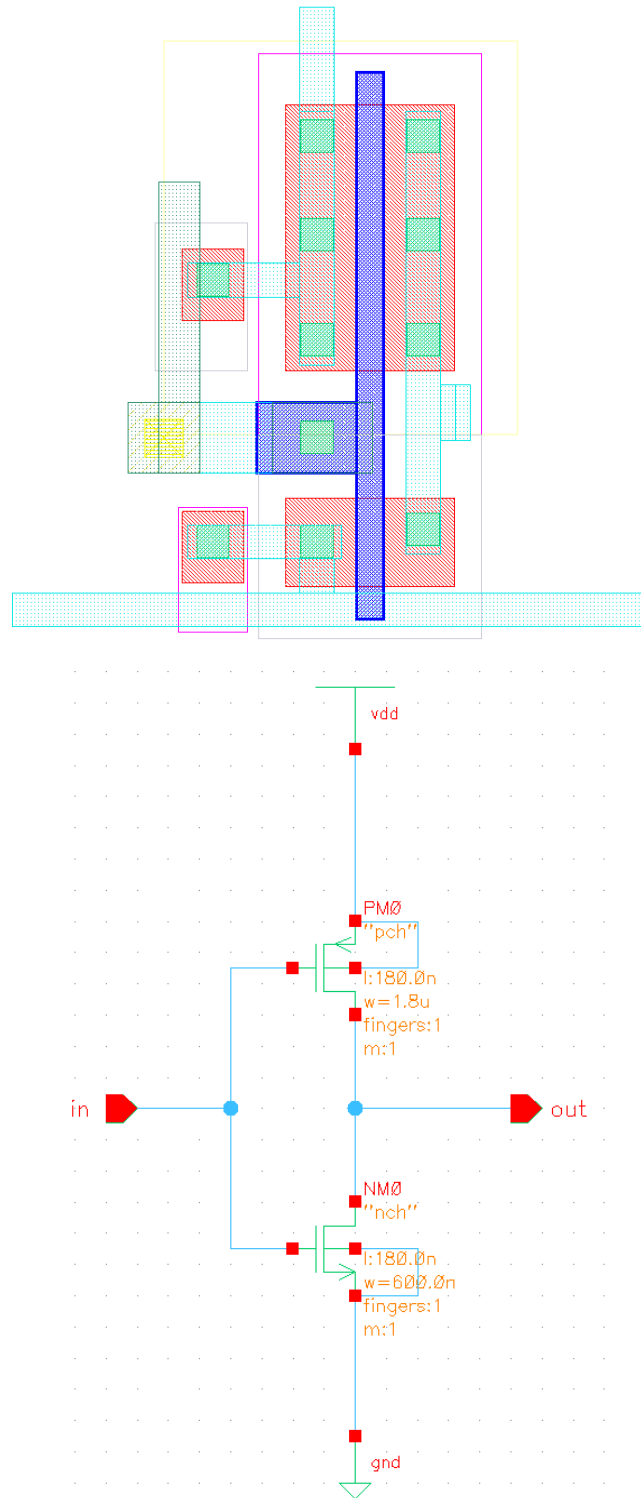


Figure 113. Inverter appearing after the OR-plane, used for needed inversion and signal restoration

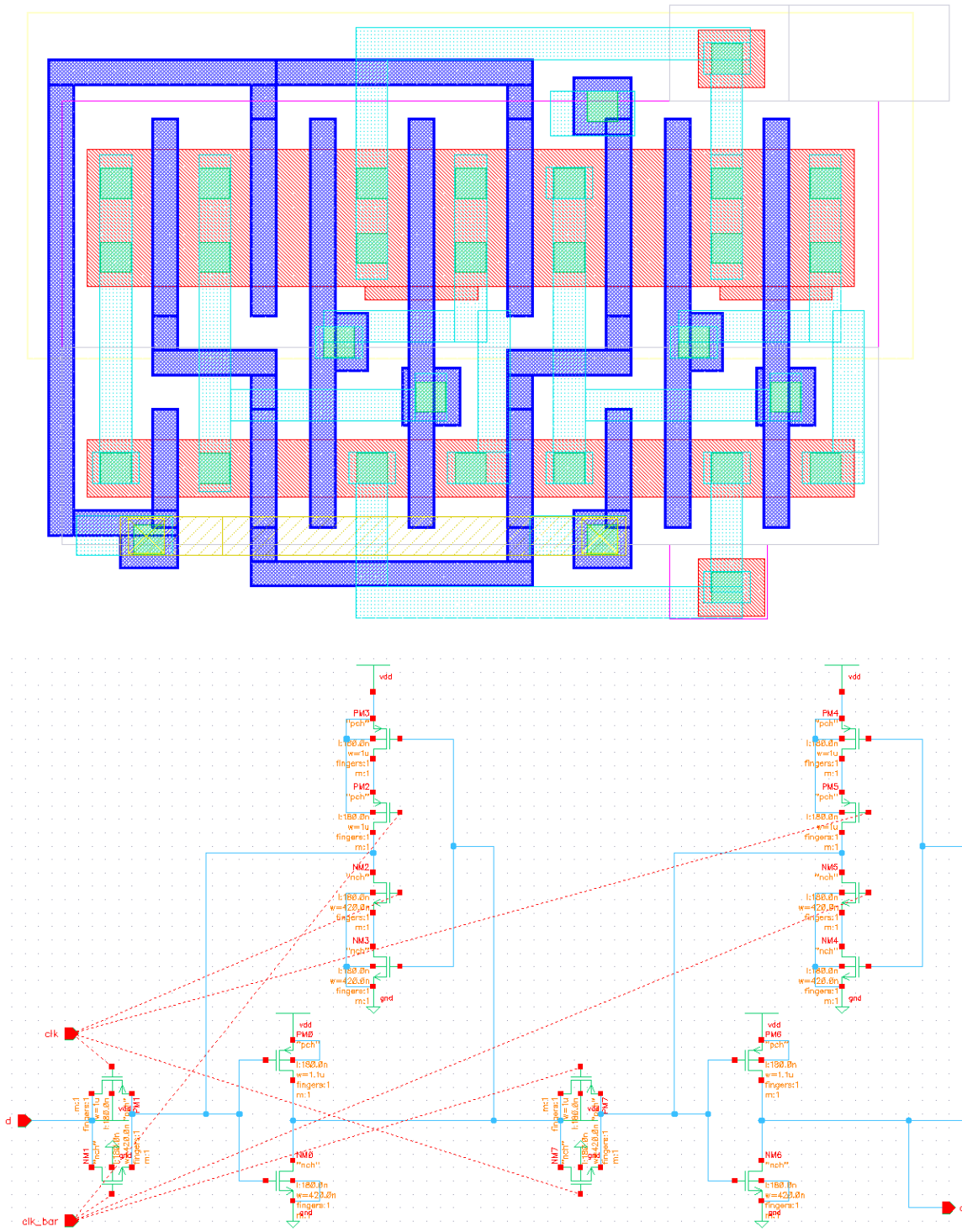


Figure 114. D-Flip-Flop used for optional registering of the PLA outputs

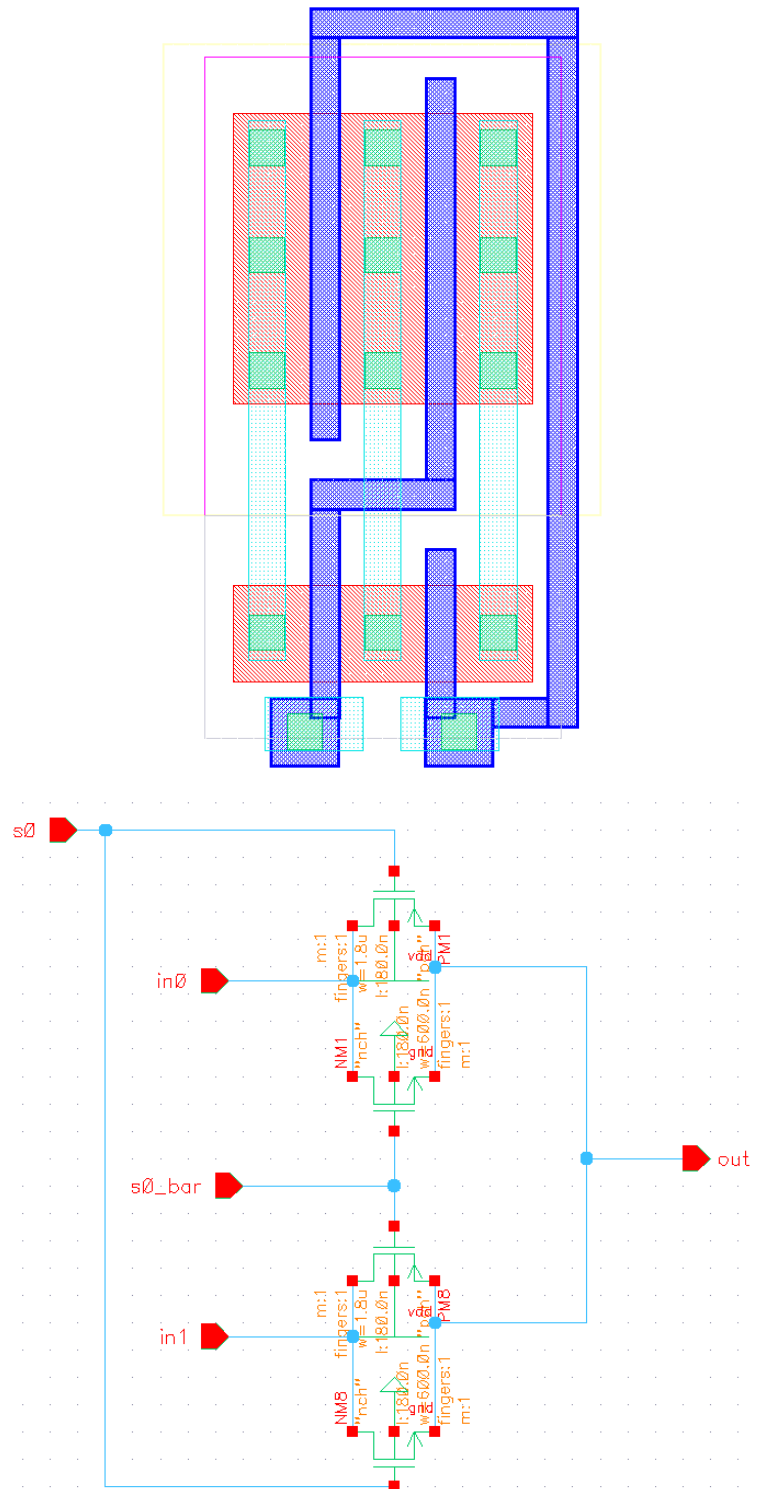


Figure 115. 2-to-1 multiplexor used for choosing the registered or unregistered PLA output

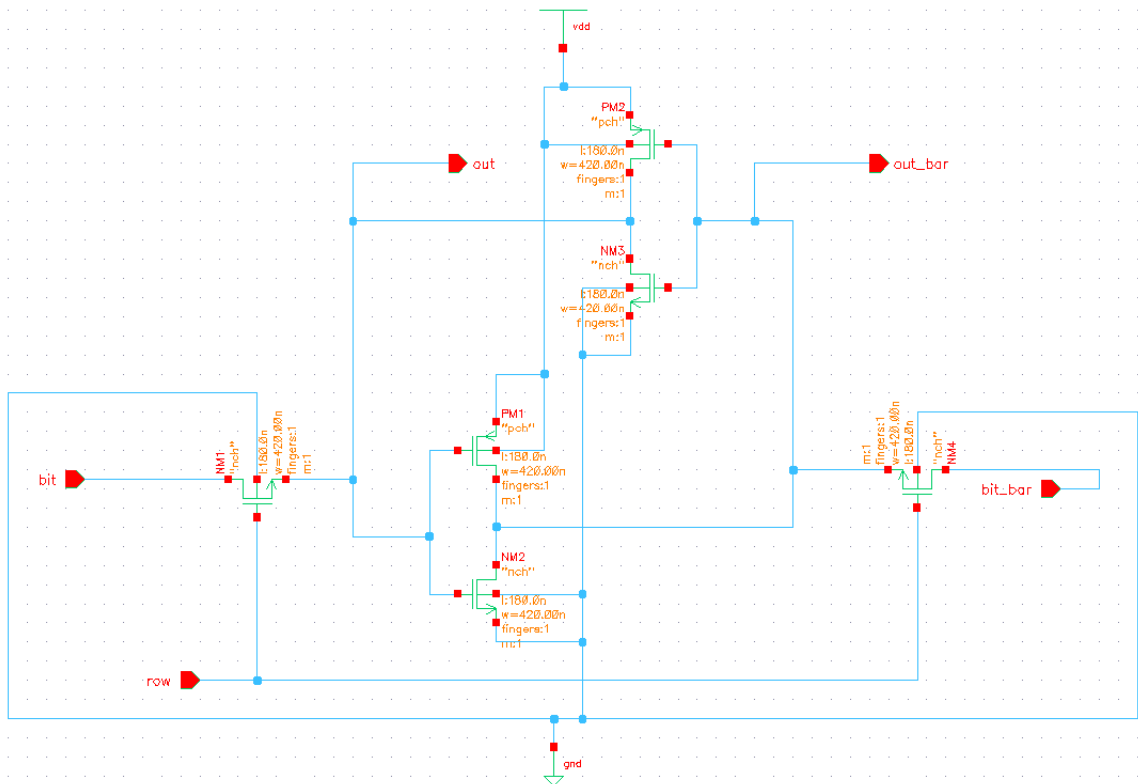
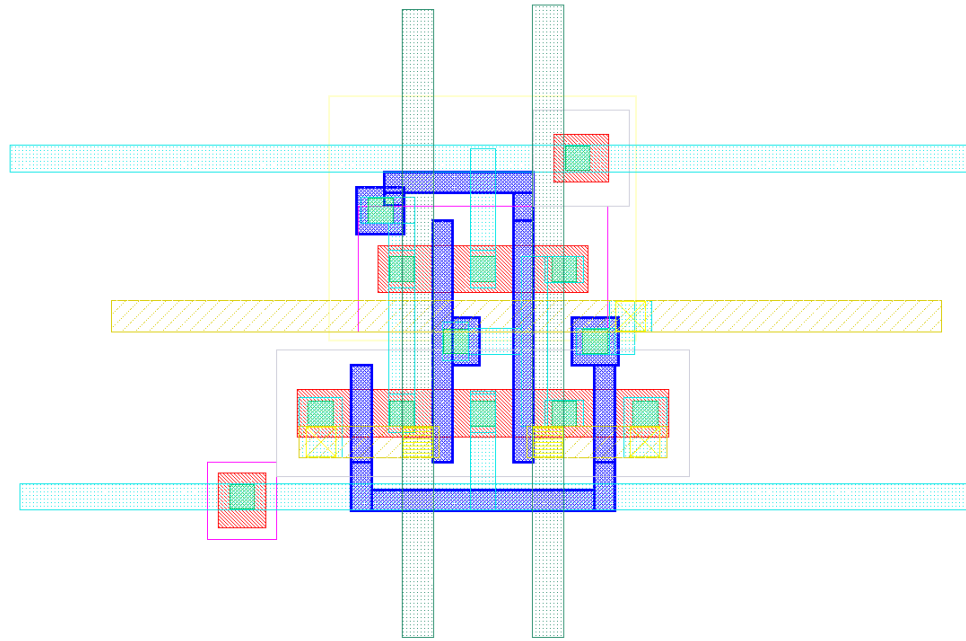


Figure 116. SRAM bit used for controlling the 2-to-1 multiplexor

Vita

Mark Holland was born in Seattle, Washington in September of 1977, and has called the city home for all of his life. He earned a Bachelor of Science in Engineering from Harvey Mudd College in 2000, and a Master of Science in Electrical Engineering from the University of Washington in 2002. In 2005 he earned a Doctor of Philosophy in Electrical Engineering from the University of Washington.