

Automatic Creation of Reconfigurable PALs/PLAs for SoC

Mark Holland, Scott Hauck

Department of Electrical Engineering
University of Washington, Seattle, WA 98195, USA
mholland@ee.washington.edu, hauck@ee.washington.edu

Abstract. Many System-on-a-Chip 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 even be used for post-fabrication modifications. Also, by catering the logic to the SoC domain, additional area/delay/power gains can be achieved over current, more general reconfigurable fabrics. This paper presents tools that automate the creation of domain specific PLAs and PALs for SoC, including an Architecture Generator for making optimized arrays and a Layout Generator for creating efficient layouts. By intelligently mapping netlists to PLA and PAL arrays, we can reduce 60%-70% of the programmable connections in the array, creating delay gains of 15%-30% over unoptimized arrays.

1 Introduction

As device scaling continues to follow Moore's Law, chip designers are finding themselves with more and more real estate to work with. This is true in the design realm of System-on-a-Chip (SoC), where individual, pre-designed subsystems (memories, processors, DSPs, etc.) are integrated together on a single piece of silicon in order to make a larger device. Moore's Law can be seen as providing us with more silicon space to use, and one solution for SoC designers is to add reconfigurable logic to their devices.

Reconfigurable logic fills a useful niche between the flexibility provided by a processor and the performance provided by custom hardware. Traditional FPGAs, however, provide this flexibility at the cost of increased area, delay, and power. As such, it would be useful to tailor the reconfigurable logic to a user specified domain in order to reduce the unneeded flexibility, thereby reducing the area, delay, and power penalties that it suffers. 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 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 that domain.

Previous work in Totem has focused on using a 1-D RaPiD array [1-4] in order to provide reconfigurable architectures for domains that use ALUs, Multipliers, RAMs,

and other coarse grained units. But many domains do not benefit from the use of coarse-grained units, and would require a finer-grained fabric. An appealing solution for these domains would be to create reconfigurable PALs and PLAs, which are efficient at representing seemingly random or non-regular logic functions. Providing SoC designers with reconfigurable PALs and PLAs will give them the ability to support many different fine-grained functions, perform run-time reconfiguration, or even perform bug fixes or other post-fabrication modifications.

2 Background

Reconfigurable PALs and PLAs have existed for many years in commercially available CPLDs, and are produced by companies including Xilinx, Altera, and Lattice. CPLDs are typically reprogrammable PALs or PLAs that are connected by fairly rich programmable routing fabrics, with other hardware added to increase the functionality of the devices. These commercial CPLDs, however, suffer from the same drawbacks as commercial FPGAs: their generality, while allowing them to be used in many domains, costs them in terms of area, delay, and power.

Additionally, many papers have been published with respect to PAL and PLA architectures. The most applicable of these was by A. Yan and S. Wilton [5]. In this paper they explore the development of “soft” or synthesizable programmable logic cores based on PLAs, which they call product term arrays. In their process they acquire the high-level requirements of a design (# of inputs, # of outputs, gate count) and then create a hardware description language (HDL) representation of a programmable core that will satisfy the requirements. This HDL description is then given to the SoC designer so that they can use the same synthesis tools in creating the programmable core that they use to create other parts of their chip.

Their soft programmable core has the advantages of easy integration into the ASIC flow, and it will allow users to closely integrate this programmable logic with other parts of the chip. The core will likely be made out of standard cells, however, whose inefficiency will cause significant performance penalties. As such, using these soft cores only makes sense if the amount of programmable logic required is small.

Our process differs from [5] in that we will create and provide domain-specific “hard” cores to be used in SoC. Our tools will intelligently create a PLA or PAL architecture that fits the specifications provided by the designer, and it will then use pre-optimized layouts to create a small, fast, low-power layout of the array that can be immediately placed onto the SoC. This will result in area, delay, and power improvements over pre-made programmable cores or soft cores, and it is automated in order to provide very fast turnaround time.

Another related work presents highly regular structures which provide ease of design and layout of PLAs for possible use in SoC [6]. Their proposal is to stack multiple PLAs in a uni-directional structure using river routing to connect them together, resulting in a structure that benefits from both high circuit regularity and predictable area and delay formulation. Their focus on circuit regularity and area/delay predictability prevent them from obtaining high performance, however. Our arrays will be tailored to the specifications of the designer, allowing us to both

better suit their exact needs and to make modifications that will result in better area, delay, and power performance.

In regards to software, much work has been done on the concept of minimizing sum-of-products style equations so that they will require smaller PLA or PAL arrays for their implementation. To date, the most successful algorithm for minimizing these equations is ESPRESSO, which was developed at Berkeley in the 1980s [7]. ESPRESSO's basic strategy is to iteratively expand its terms (in order to encompass and remove other terms) and then reduce its terms (to prepare for a new, different expansion). This expand and reduce methodology results in a final equation that has a near-optimal number of product terms. ESPRESSO also reduces the number of literals in the equation, which is equivalent to minimizing the number of connections that need to be made in the PLA or PAL array.

3 Approach – Tool Flow

We envision the tool flow to be as follows. The input from the customer will be a specification of the target domain, most likely containing a set of netlists that the architecture will need to support. These netlists will first be processed by ESPRESSO in order to minimize the number of product terms and literals that they contain.

The resulting minimized netlists will be fed into the Architecture Generator, which attempts to create the smallest single PLA or PAL array that is capable of being configured to every netlist. Only one netlist must be supported at any given time. The Architecture Generator will then output a description of this array.

The array specification from the Architecture Generator is fed to the Layout Generator, which creates a full layout of the array. This layout includes the PAL/PLA array as well as all the remaining logic that is necessary for programming the array.

3.1 Architecture Generator

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

The goal of the Architecture Generator is to map all the netlists 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 netlists. 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 netlist. Figure 1 displays the problem that we face when trying to minimize the number of programmable connections. In this example we are trying to fit two netlists (one grey and one black) onto the same array. A random mapping of the product terms is shown to require 23 programmable

connections, while an intelligent mapping is shown to require only 12 programmable connections - a 48% reduction.

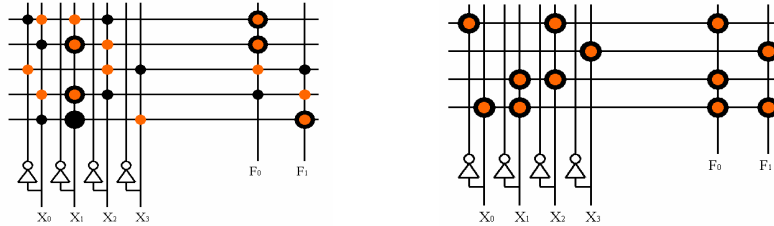


Fig. 1. On the left, two netlists are mapped randomly. This requires 23 programmable bits. On the right, they are mapped intelligently, requiring only 12 programming bits

For our application, simulated annealing has proven to be very successful at mapping netlists to an array. The algorithm’s goal is to minimize the number of programmable connections. We define a basic “move” as being the swapping of two product term rows within a netlist (we will introduce more complicated moves later), and the “cost” of a mapping is the number of programmable bits that it requires. For our annealing we use the temperature schedules published in [8].

The development of a cost function requires serious consideration, as it will be the only way in which the annealer can measure netlist placements. At first glance, the cost of an array location is ideally either a 1 (programmable bit present) or a 0 (programmable bit not present). Using a simple 1/0 cost function, however, would hide a lot of useful information from the annealer. The degree to which a programmable bit is required (how many netlists are using it) is also useful information, as it can tell the annealer how close we are to removing a programmable connection.

Figure 2 displays this notion. In this example, we have one programmable connection used by two netlists and another connection used by five netlists. 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.

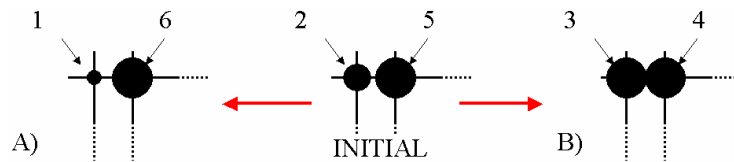


Fig. 2. A move to A) puts us closer to removing a programming bit than a move to B)

The cost function that we developed captures this subtlety by adding diminishing costs to each netlist that uses a programmable connection. If only one netlist is using a connection the cost is 1; if two netlists use a connection it costs 1.5; three netlists is 1.75, then 1.875, 1.9375, and so on. Referring back to Figure 2 and using this cost

function, moving to figure A) is now a cost of $-.45$ (a good move) while moving to figure B) is a cost of $.19$, which is a bad move. Mathematically, the cost function is:

$$\text{COST} = 2 - .5(x - 1)$$

where x is the number of netlists that are using a position. As seen, each additional netlist 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.

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 netlist mappings. Considering this, we are presented with a need for four annealing scenarios: using a PLA with fixed IO positions, using a PLA with variable IO positions, using a PAL with fixed IO positions, and using a PAL with variable IO positions.

The differences between the annealing scenarios are shown in Figure 3. Given a PLA with fixed IO positions, the only moves that we can make are swaps of product terms within a netlist (A). Given variable IO positions (B), however, we can also make swaps between the inputs of a netlist or between the outputs of a netlist.

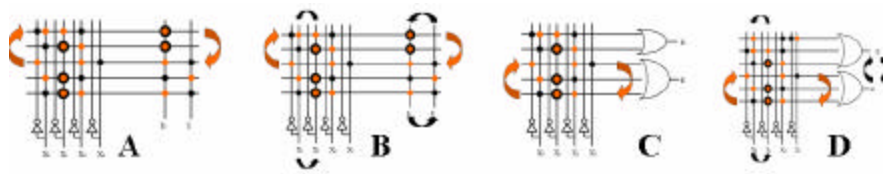


Fig. 3. Allowed annealing moves for the four scenarios

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

For the PLA with variable IO positions, 50% of the moves are product term swaps and 50% are IO swaps, with the ratio of input to output swaps equal to the ratio of inputs to outputs. For the PAL with variable IO 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.

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.

3.2 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. The Layout Generator runs in Cadence's LayoutPlus environment, and uses a SKILL routine that was written by Shawn Phillips. We are currently designing in the TSMC .18-micron process.

Figure 4 shows a PLA that our Layout Generator created (a very small array has been shown for clarity, 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.

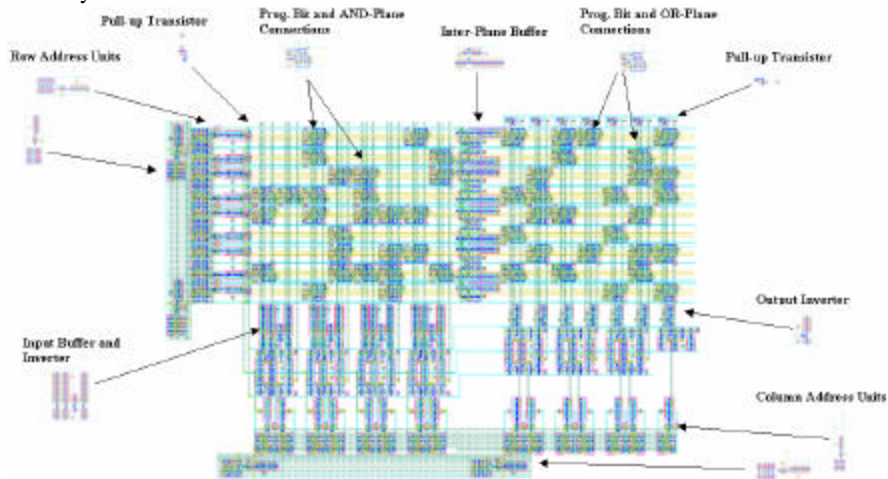


Fig. 4. The PLA is created by tiling pre-made, optimized layout cells

Currently, the PLAs and PALs are implemented using a pseudo-nMOS logic style. 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.

4 Methodology

The use of PALs and PLAs restricts us to the use of .pla format netlists. The first source of netlists is the ESPRESSO suite (the same netlists on which the ESPRESSO algorithm was tested). A second set of netlists comes from the benchmark suite compiled by the Logic Synthesis Workshop of 1993 (LGSynth93). As a whole, these netlists are commonly used in research on programmable logic arrays. The netlists

are generally fairly small, but this suits our needs as we are currently only using single arrays to support them.

From these sources we found netlists with similar numbers of inputs, outputs, and product terms and we put them together in groups (1 through 4). The netlists are grouped according to size, as this will be a factor in how well our algorithms perform.

5 Results

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, the best cost function improvement that our annealer can obtain is actually bounded. For a two netlists anneal, the cost function can never improve more than 25% and the number of programming bits required can never improve more than 50%. A three netlists anneal has a maximum cost function improvement of 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 netlists. 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.

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 netlist mapped with itself: it is simply the number of connections in the netlist, as all the connections from the first netlist should map to the same locations as the connections from the second netlist. This can be done with any quantity of the same netlist, 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.

We applied this self-mapping test to five random netlists using each of the four algorithms: PLA-fixed, PLA-variable, PAL-fixed, and PAL-variable. The results showed that when two netlists are mapped with themselves using the PLA-fixed algorithm that the final mapping is always optimal. The PLA-variable had difficulty with only one netlists, *shift*, which was 15.21% from optimal. Note that for this example the random placement was 92.49% worse than optimal, so our algorithm still showed major gains.

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* netlist was only able to get 13.28% from optimal (vs. 92.23% from optimal for a random placement). These near optimal results give us confidence that our annealing algorithms should return high quality mappings for arbitrary netlist mappings as well.

Table 1 shows the results of running the PLA-fixed and PLA-variable algorithms on the different netlist groups that we created. The reduction in bit cost is the difference in the number of programmable connections needed between a random mapping of the netlists and a mapping performed by the specified algorithm. In the table, all possible 2 netlist mappings were run for each specific group and the results were then averaged. The same was done for all possible 3 netlist mappings, 4 netlist, etc., up to the number of netlists in the group.

Table 1. Average improvement in programming bits for PLA -Fixed and PLA- Variable algorithms over random placement as a function of netlist count.

Group	# Netlists	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 1. 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/delay/power results from our reconfigurable arrays by supplying external hardware to support input and output permutations.

Another thing to notice is that the reduction always increases as the number of netlists being mapped increases. This, too, is as we would expect, as adding more netlists 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 netlist, it only says that we reduce a greater number of connections from a random mapping.

Table 2 shows the results of running the PAL-fixed and PAL-variable algorithms on netlist Group 3. We see the same results from the PAL algorithms as from the PLA algorithms: increasing the number of netlists in a mapping increases the resulting bit reduction, and allowing input and output permutations always results in a better mapping than requiring fixed input/output locations.

Table 2. Average cost improvement for PAL-Fixed and PAL- Variable algorithms over random placement as a function of netlist count.

Group	# Netlists	PAL-Fixed	PAL-Var.
3	2	6.56%	12.24%
	3	12.34%	20.68%
	4	16.77%	30.99%

Another important concept is how well netlists match each other ... higher reductions will be possible when the netlists being mapped have similar sizes or a similar number of connections. With regards to array size, any netlists that are far

larger than another netlist 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 netlists, resulting in poor reduction. Results showed that arrays of similar sizes showed better reductions than arrays of disparate sizes.

Mapping netlists with a similar number of connections also results in better reductions. If netlist A has far more connections than netlist B then the total number of connections needed will be dominated by netlist A: even if we map all of the connections from netlist B onto locations used by netlist 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 netlists being mapped will allow a higher percentage of the overall programmable connections to be removed. This concept was also verified by experimentation.

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 original 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 and power gains.

We used hspice to develop delay models of both the PLA and PAL arrays that we create. Table 3 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 (netlists randomly placed and unneeded connections removed).

Table 3. Reductions obtained in number of programmable bits and delay for PLA algorithms

Netlists	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	3709	3165	2998	3620	3149	2901	2905	19936	8586	8155	7739	8505	7215	6846	6760
alu2, f51m	2156	672	557	538	1708	1073	916	908	2220	593	544	486	1627	1013	914	866
ti_xparc	42418	9807	9452	8207	5343	4573	4561	4512	269686	55952	55548	51610	33696	25974	25715	25409
b2_shift, b10	5830	2882	2510	2270	2329	2072	1999	1937	37620	10111	9627	9098	10557	6620	6581	6544
newcplal_tms, m2	1598	987	862	779	1288	1236	1216	1208	6984	3429	2998	2527	3993	3885	3831	3278
gary, b10, in2, dist	6664	3460	2658	2026	2780	2665	2610	2569	15010	5695	4852	3298	4800	3690	3392	3190
newcplal_tms, m2, exp	2124	1304	1072	966	1459	1412	1323	1372	7218	3936	3276	2679	4095	3994	3940	3406
gary_shift, in2, b2, dist	7480	4435	3516	2960	2785	2729	2646	2615	39216	12994	11437	9751	10887	7872	7289	6741
b2_shift, b10, table5.pla, misex3c.pla, table3.pla	18321	6608	4914	4521	4015	3527	3009	3118	92796	21047	16987	13453	15692	10620	9514	8513
Average Improvement Over Full	-	53.5%	61.4%	65.8%	-	11.0%	16.0%	16.1%	-	65.0%	68.9%	73.5%	-	22.4%	26.0%	31.3%
Average Improvement Over Random	-	-	16.5%	25.6%	-	-	6.0%	6.2%	-	-	10.5%	23.0%	-	-	5.0%	11.4%

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

The PLA-Variable algorithm does significantly 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.

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 Conclusions

In this paper we have presented a complete tool flow for creating domain specific PALs and PLAs for System-on-a-Chip. We have presented an Architecture Generator that, given netlists as an input, maps the netlists onto a PLA or PAL array of minimal size which uses a near-optimal number of programmable connections. Results show that higher reductions are possible if the inputs and outputs are permutable, suggesting that an SoC designer would want to have crossbars at the inputs and outputs of our arrays. Also, as the number of netlists being mapped to an array increases, the bit reduction between a random mapping and an intelligent mapping increases.

We also presented a Layout Generator that takes PLA or PAL descriptions from the Architecture Generator and successfully creates optimized layouts by tiling pre-made layout units.

References

- [1] K. Compton, S. Hauck, "Totem: Custom Reconfigurable Array Generation", IEEE Symposium on FPGAs for Custom Computing Machines Conference, 2001.
- [2] K. Compton, A. Sharma, S. Phillips, S. Hauck, "Flexible Routing Architecture Generation for Domain-Specific Reconfigurable Subsystems", International Symposium on Field Programmable Logic and Applications, 2002.
- [3] S. Phillips, S. Hauck, "Automatic Layout of Domain-Specific Reconfigurable Subsystems for System-on-a-Chip", ACM/SIGDA Symposium on Field-Programmable Gate Arrays, 2002.
- [4] A. Sharma, "Development of a Place and Route Tool for the RaPiD Architecture", Master's Thesis, University of Washington, 2001.
- [5] A. Yan, S. Wilton, "Product Term Embedded Synthesizable Logic Cores", IEEE International Conference on Field-Programmable Technology, 2003.
- [6] F. Mo, R. K. Brayton, "River PLAs: A Regular Circuit Structure", DAC, 2002.
- [7] R. K. Brayton, G. D. Hachtel, C. T. McMullen, A. L. Sangiovanni-Vincentelli, Logic Minimization Algorithms for VLSI Synthesis, Kluwer Academic Publishers, Boston, 1984.
- [8] V. Betz, J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research", International Workshop on Field Programmable Logic and Applications, 1997.