Technology Mapping and Packing for Coarse-Grained, Anti-Fuse Based FPGAs[*]

Chang Woo Kang, Ali Iranli, and Massoud Pedram
Department of Electrical Engineering – Systems
University of Southern California, Los Angeles, CA 90089
{ckang, iranli, pedram}@usc.edu

Abstract - We present a new synthesis flow for anti-fuse based FPGAs with multiple-output logic cells. The flow consists of two steps: mapping and packing. The mapper finds mapping solutions using a dynamic programming-based approach that finds the best match at each node of the decomposed target circuit. After this mapping step is completed, the resulting netlist of cells is optimally packed into net list of logic cells by using a multi-dimensional coin change problem formulation which is again solved by a dynamic programming based approach. Experimental results for Quicklogic's pASIC3 logic family are provided to assess the effectiveness of the proposed mapping and packing techniques.

I. Introduction

Fast time-to-market is the pivotal success factor in today's ever-changing electronics market. Field programmable gate arrays (FPGAs) can create unique advantages over application specific integrated circuits (ASIC) because of quick and cost-effective validation of products. An extensive survey of existing SRAM based FPGA mapping techniques is given by Cong and Ding [1].

In this paper, we present a synthesis technique for anti-fuse FPGAs and its basic logic cell has multiple outputs as shown in Figure 1, which has been introduced by QuickLogic [2]. The proposed algorithm consists of the following steps. First, we extract three single-output base gates from the Quicklogic multiple-output logic cell. These base gates can be implemented (individually or concurrently) in the logic cell by setting the correct bit values for the input multiplexers in Figure 1. We generate a standard gate library where each primitive cell in the library is a personalization of one these base gates by using stuck-at (constant assignment) and bridging (shorting fault). Next, we map the target circuit to this gate library by employing a dynamic programming paradigm such as the one employed in well-known tree covering algorithms [3][4]. Finally, we assign these single-output functions to multiple-output logic cells using an efficient packing algorithm. We used a dynamic programming technique to find an optimal packing solution so that the minimum number of logic cells is used to realize the circuit. This packing problem is a multi-dimensional extension of the coin-change problem.

This paper is organized as follows. The library generation process is explained in section II. Technology mapping and the algorithm for packing the primitive cells into the target logic cells are presented in section III, while experimental results and conclusion are given in section IV.

II. Cell Library Construction

The complete pASIC3 logic cell consists of two 6-input AND gates, four 2-input AND gates, six 2:1 multiplexers and one D flip-flop with asynchronous set and reset controls. Since all connections within the cell are hard-wired, the various functions are available in parallel. Thus, very wide, complex functions are implemented with the same cell speed (about 2ns) as the much smaller "fragment" functions. Related and unrelated functions can be packed into the same logic cell, increasing effective density and gate utilization.

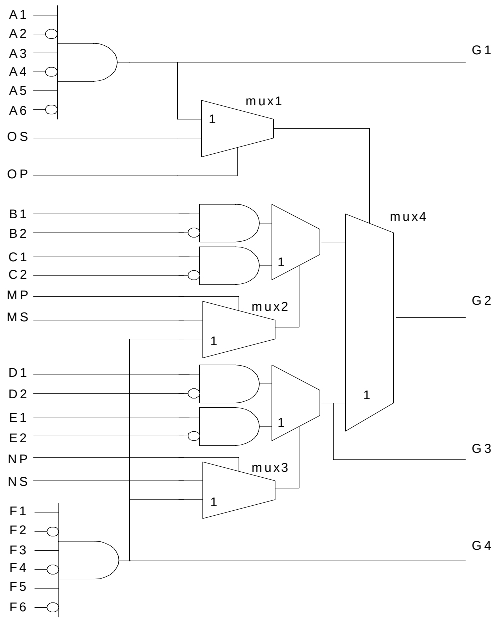


Figure 1: QuickLogic anti-fuse configurable logic cell.

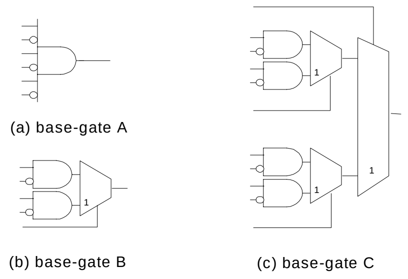


Figure 2: pASIC3 Base gates derived from the reconfigurable logic cell.

The number of gates that may be generated from the pASIC3 logic cell by using assigning 0 or 1 to inputs (i.e., inputs connected to either VDD or GND levels) is quite large. Therefore, we break the pASIC3 logic cell into manageable sub-blocks at the expense of not exploiting the full flexibility/programmability of the larger block. By appropriately connecting the control inputs of the four multiplexers (cf. **mux1** through **mux4** in Figure 1) to zero or one logic levels, three base logic gates (A, B, and C) can be obtained as shown in Figure 2. Through experimentation with a large number of test bench circuits, we were able to identify 205 cells as the most useful primitive cells. Note that some library cells may be generated from different base gates. Figure 3 shows

[*] This research was supported in part by NSF under grant no. 9988441.

the set relationship (Venn's diagram) for the three different sets of primitive cells. Notice that some of the primitive cells in the library can be generated from two or more single-output base gates, while others can only be derived from one of the base gates. Furthermore, there is a fixed number of ways to embed (pack) a maximal number of these cells in the pASIC3 logic cell. In fact, there are 37 different cases of completely utilizing the logic cell except for the multiplexers of course. A number of these cases (but not all) are shown in Table 1. In this table, LC_i refers to the i_{th} combination of completely utilizing the pASIC3 logic cell with the *primitive cell types*. Let $C_{i,Sj}$ denote the number of primitive cells from set S_j in the i_{th} combination.

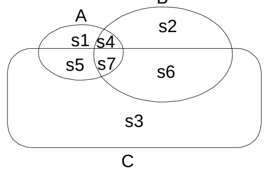


Figure 3: Venn's diagram for the set of logic cells that can be personalized from the base gates.

Table 1: Full packing solutions of pASIC3 logic cell (for definitions of sets S_1 to S_7 see Figure 3).

LC_i	Combinations of primitive cells
1	$2S_5 + 2S_7$
2	$2S_4 + 2S_5$
3	$2S_5 + 2S_6$
…	…
i	$\sum_{i=1}^{7} C_{i,S_i} \times S_i$
…	…
35	$S_3 + 2S_7$
36	$S_3 + 2S_4$
37	$S_3 + S_4 + S_7$

The cost of a library cell must be assigned carefully. There are three factors that determine the cost of a library cell: *freedom* (f), *coverage* (c), and *space usage* (s). The freedom parameter captures the total number of places in the base gates that the library cell can fit. The coverage parameter accounts for the complexity of the logic that the library cell realizes. It is measured in terms of the number of literals in the minimal factored form representation of the logic function. The space usage parameter represents the amount of space inside a pASIC3 logic cell that is used up by the library cell. The cost of each library cell is calculated by the following equation:

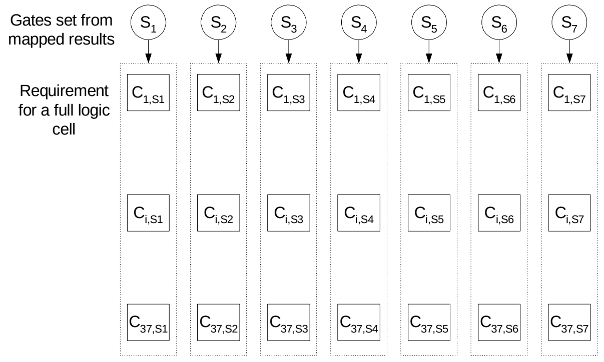
$$COST = \frac{s}{f \cdot c}$$

A simple inverter does not have the lowest cost because it consumes one out of four slots. However, it only operates as an inverter. In other words, library cells with higher freedom, larger coverage, and lower space usage are much more preferable.

III. Technology Mapping and Cell Packing

A. *Technology mapping*

We perform minimum-area technology mapping for a network with the cell library generated as described in section II. We use the SIS mapper [6] to generate the mapped netlist. Next, we need to minimize the number of logic cells required to implement the netlist in pASIC3 FPGAs. This problem is formulated as a cell packing problem and solved by a dynamic programming approach, as explained next.



$$\textit{Minimize} \quad \sum_{i=1}^{37} n_i$$

$$\textit{s.t.} \quad \bigcap_{j=1}^{7} \left\{ \sum_{i=1}^{37} n_i \times C_{i,S_j} \geq |S_j| \right\} = 1$$

Figure 4: Multi-dimensional coin change problem.

B. *Determining the minimum number of pASIC3 logic cells*

The packing problem can be stated as the following: given 37 cases of filling a pASIC3 logic cell by cells derived from the base gate types and the netlist of cells generated by the mapper, find the minimum number of logic cells to cover all cells in the netlist. This is the same problem as the well-known *coin change* problem as defined next.

To solve the cell-packing problem, we must extend the coin change problem. Figure 4 presents the general problem formulation. n_i is the number of pASIC3 logic cells, which use the i_{th} combination of primitive cell types to fill a pASIC3 logic cell. There are 37 cases of legal combinations of primitive cells to fill a pASIC3 logic cell. Let $C_{i,Sj}$ denote the number of primitive cells from set S_j in the i_{th} legal combination (cf. Table 1).

The recurrence equation for this problem can be as follows:

$$count\left(|S_1|,...,|S_7|\right) = \begin{cases} 0 & \textit{if } \forall |S_j| \leq 0 \\ \min\limits_{\forall i}\left(count\left(|S_1| - C_{i,S_1},...,|S_7| - C_{i,S_7}\right) + 1\right) & \textit{otherwise} \end{cases}$$

where $|S_i|$ is the number of remaining gates in the set S_i. The algorithm can be implemented by table lookup as shown in Figure 5. It can fill up two, seven-dimensional tables from bottom to the top. The *count* array is used to store the minimum number of pASIC3 logic cells for each sub-problem, whereas the *comp* array is used to store the selected combination out of the 37 choices for the optimal selection for the sub-problem. In lines 16 and 17, the minimum number of logic cells for a sub-problem and the selected combination will be stored, respectively. To construct the exact solution, the algorithm steps back from top to bottom depending on values in those tables. The complexity is polynomial in $O\left(M \prod_{i=1}^{7} |S_i|\right)$ where M is the number of combinations of primitive cell types needed to fill a pASIC3 logic cell.

To reduce the space complexity, the circuit is partitioned beforehand. More precisely, a linear ordering of cells is constructed by using a *level-first search* of the target circuit (this is a topological ordering starting from the leaf nodes of the underlying circuit graph). Next, the list is divided into several sub-lists. Each sub-list is then processed separately.

```
Algorithm Pack-cells(|$S_1$|,…,|$S_7$|, $C_{1,s1}$,…, $C_{37,s7}$)
begin
  count[0,…,0] = 0
  for s1 = 1,…, |$S_1$|
   for s2 = 1,…, |$S_2$|
    for s3 = 1,…, |$S_3$|
     for s4 = 1,…, |$S_4$|
      for s5 = 1,…, |$S_5$|
       for s6 = 1,…, |$S_6$|
         for s7 = 1,…, |$S_7$|
           min = ∞
           for i = 1,…,37
            if count[$s_1$-$C_{i,s1}$,…,$s_7$-$C_{i,s7}$] +1 < min
            then
              min = count[$s_1$-$C_{i,s1}$,…,  $s_7$-$C_{i,s7}$] +1
              which = i
            count[s1,…,s7] = min
            comp[s1,…,s7] = which
end
```

Figure 5: Pseudo code for packing cells.

By running the dynamic programming-based cell packing, the required pASIC3 logic cells are generated. Each pASIC3 logic cell has been assigned types of primitive cells for each of its base gates from the *count* and *comp* tables. Then, we collect these primitive cells into a netlist of pASIC3 logic cells that represents the same original circuit.

IV. Implementation and Experimental Results

For the simulation, we developed a cell packer based on dynamic programming, and integrated it into the SIS environment [6]. We compared our packing results with those obtained by a *greedy algorithm*. The greedy algorithm packs mapped primitive cells into pASIC3 logic cells as follows. First, the mapped cells are ordered by the level-first search strategy. For each full packing solution of pASIC3 logic cell (there are 37 such logic cell compositions as reported in Table 1), the greedy algorithm tries to fill it as much as possible starting from the top of the list. Next, it selects the logic cell combination that has the best space utilization where utilization is defined as the ratio of filled space to the total space of the pASIC3 logic cell. The selected combination is added to the final pASIC3 netlist, and the corresponding mapped cells are removed from the list. This procedure is repeated until the list becomes empty at which point the final pASIC3 mapping solution has been generated. Table 2 shows the simulation results with various circuits from 91 MCNC IWLS benchmarks. The average improvement by using the dynamic programming-based technique over the greedy algorithm is 26.9%. Note that both algorithms use the results of the SIS mapper as the initial solution. Our dynamic programming algorithm outperformed the greedy algorithm. For example, it achieved 100% cell utilization for alu2 and C432, while the greedy algorithm resulted in less than 70% cell utilization.

V. Conclusions

In this paper, we presented a logic synthesis technique to generate a mapping solution for anti-fuse FPGAs with multiple-output logic cells. We first generated library gates from base gates, and then BMX mapped gates for a target circuit. A dynamic programming technique, an extension of a coin-changing problem, was used to find the minimum number of logic cells to map the circuit into logic cells. Simulation results show that the packing algorithm provides 27% improvement over a greedy algorithm.

Table 2: Performance comparison between optimal packing and greedy packing

Circuits	Primitive cell count	Greedy packing			Dynamic programming based packing			Packing improvement	
		Number of logic cells	Cell utilization (%)	CPU time (sec)	Number of logic cells	Cell utilization (%)	CPU time (sec)	Number of logic cells (%)	Cell utilization (%)
alu2	193	76	65.87	0.08	51	100	50.27	32.9	34.1
alu4	377	150	65.11	0.34	101	98.43	960.59	32.7	33.9
apex6	349	154	59.86	0.37	117	80.23	306.24	24.0	25.4
dalu	471	194	63.39	0.56	138	90.75	143.81	28.9	30.1
C1355	210	83	64.81	0.11	58	93.75	1.37	30.1	30.9
C1908	213	96	58.84	0.13	74	77.74	20.54	22.9	24.3
C432	108	45	65.85	0.03	31	100	13.36	31.1	34.2
C499	210	83	64.81	0.11	58	93.75	1.37	30.1	30.9
C3540*	657	268	64.22	1.2	191	91.89	56.21	28.7	30.1
C880	214	92	62.76	0.12	64	93.45	45.95	30.4	32.8
C5315*	764	333	59.97	1.94	256	79.09	42.97	23.1	24.2
C6288*	1457	664	55.19	13.11	593	61.84	19.14	10.7	10.8
C7552*	1052	413	64.94	3.70	312	86.51	86.06	24.5	24.9
Average								26.9	28.2

*Packing algorithm based on dynamic programming used segmented lists not to exceed available memory.

References

[1] J. Cong and Y. Ding, “Combinational logic synthesis for LUT-based field-programmable gate arrays,” *ACM Transactions on Des. Automat. Electron. System*, April, pp. 145 – 204, 1996.

[2] pASIC3 FPGA Family Datasheet, QuickLogic Corporation (http://www.quicklogic.com).

[3] A. Aho and S. Johnson, “Optimal code generation for expression trees,” *Journal of ACM*, vol. 23, no. 3, pp. 488 – 501, June 1976.

[4] K. Keutzer, “Dagon: technology binding and local optimization by dag matching, “in *Proc. Design Automation Conference*, 1987, pp. 341 – 347.

[5] J. Cong, M. Romesis, “Performance-driven multi-level clustering application to hierarchical FPGA mapping,” in *Proc. Design Automation Conference*, June 2001, pp. 389 – 394.

[6] E.M. Sentovich, et al., *SIS: A system for sequential circuit synthesis*, 1992, Electronics Research Laboratory, College of Engineering, University of California, Berkeley