Amap: a Technology Mapper for Selector-based Field-Programmable Gate Arrays

Kevin Karplus*

Baskin Center for
Computer Engineering & Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064 USA

ABSTRACT

This paper presents two algorithms for doing mapping from multi-level logic to selector-based field-programmable gate arrays, such as the Actel chip.

The gate counts and CPU time are compared with two previous mappers for these architectures: misII and mis-pga. The Amap algorithm use 6% fewer cells than misII and only about 8% more cells than the best achieved by mis-pga, and is at least 25 times as fast as misII and at least 586 times as fast as mis-pga. The XAmap algorithm is slightly slower, and not quite as effective.

1 Introduction

Two algorithms are presented in this paper: Amap and XAmap, both for mapping combinational logic to selector-based gate arrays, such as the ones produced by Actel. The Amap algorithm tries to match the selector structure of the gate array with a selector-based representation of the function. The XAmap algorithm uses Xmap to map to arbitrary three-input functions [7], then replaces those functions with Actel cells that implement them.

The algorithms are based on a multiply-rooted if-then-else directed acyclic graph representation of the functions [4]. In order to compare the new mapping algorithms fairly with existing mappers, they have been run on the output of the misII logic minimizer [1], which is in Berkeley Logic Interchange Format (BLIF). The conversion from BLIF done is exactly the same as is done for the Xmap algorithm [7].

The gates found by Amap and XAmap are (possibly overlapping) sub-DAGs of the if-then-else DAG for the entire circuit. Because of this *direct mapping*, the mappers preserve any path-delay-fault testability of the underlying DAG [6].

2 The Actel cell

The Actel chip is designed to have many copies of a small, versatile cell, as is appropriate when the cost of the many connection points is low. Actel's basic cell (illustrated in Figure 2.1) uses selectors as the basic gates, and can be configured for either combinational logic or storage [3]. We will consider only combinational logic in this paper.

Each input of the cell can be connected to 0, 1, or a wire from another cell or chip input. The cell is quite flexible—the designers claim that all two- and three-input functions can be implemented in a single cell, as well as several functions with more inputs [3, page 753]. All the two-input functions

*This research was funded by NSF grant MIP-8903555.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

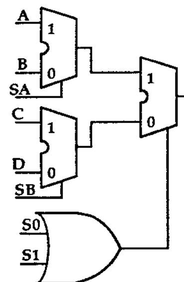

Figure 2.1: Basic cell used in Actel chips field-programmable-gate-array chips.

are trivially implementable, but only 213 of the 256 different three-input functions can be implemented in a single cell (determined by generating the 5^8 ways of connecting three inputs to an Actel cell, converting to canonical form, and tabulating the results). The missing 43 functions are given in the technical report [5, Table 4.1]. If one of the variables is provided dual-rail, then all three-input functions can be implemented directly from the truth table—connect $S_0 = a$, $S_1 = 0$, $S_a = S_b = b$, and connect A, B, C, and D to 0, 1, c, or c' as needed.

3 Amap algorithm

The Amap algorithm consists of two passes. In the first pass, Amap does some minor, local manipulation of the if-then-else DAG to make the second pass work better. The second pass does the mapping with a top-down recursive function that is called once for each output. A node of the DAG is passed to the function, which generates an Actel cell whose output corresponds to that node, and calls itself on the inputs. Most of the complexity of the function comes in deciding how much of the DAG to cover.

3.1 Local manipulation

The mapper does a traversal of the DAG, looking for triples that represent two-input commutative functions. If exactly one literal is an input, the triple is commuted, if necessary, to make the literal be in the if-part. Because the if-parts will later be matched to the selector inputs of the Actel cells, this reorganization tends to use the selectors more efficiently, resulting in about 0.5% fewer cells. This commute order may not be the final one used, as the mapping algorithm frequently checks two-input triples to see if the commuted form has become cheaper.

The first pass also assigns a preferred polarity for each node. The concepts of commute order and polarity are important, but the first pass has little effect on the success of the mapper, causing only about a 1% reduction in the number of cells used. The second pass ends up making most of the decisions independent of the first pass.

©1991 ACM 0-89791-395-7/91/0006/0244 $1.50

3.2 Matching Actel cells to the DAG

After the commute order and initial polarity assignments are done, the if-then-else DAG has to be covered with the selectors of Actel cells. The covering is done in a single pass over the DAG. A recursive function is called for each root of the DAG, that is, for each principal output of the circuit. The function chooses nodes in the DAG to be the inputs for an Actel cell, then calls itself recursively to map those inputs.

The recursion stops when the node that Amap is trying to map is a literal, or is already implemented in either polarity. In both cases, Amap either creates no cells, or creates a cell to act as a simple inverter, depending on what polarity is needed.

For non-trivial nodes, the function has to decide how much of the Actel cell to use. Being greedy, stuffing as much as possible into the cell, results in considerable duplication of functions, as nodes that could have been shared get hidden inside Actel cells. If too little is stuffed into the Actel cell, then the cells are used inefficiently, and more of them are needed. For example, if we never use more than one input to the OR-gate, we'll need about 5% more cells than if we use both inputs wisely. Sections 3.2.1, 3.2.2, and 3.2.3 describe the heuristics used to choose how much of the DAG to include in the Actel cell.

To simplify the discussion of the heuristics, let's define a *free* OR as a node that is already implemented in either polarity, or is the OR or NOR of two signals that exist in the correct polarity. Note that a free OR can be implemented as the control input for the output selector with no additional Actel cells.

Similarly, let's define a *free selector* as a node in the if-then-else DAG that exists in either polarity, or the if-branch exists in either polarity and the then- and else-branches exist in the appropriate polarities. Note that a free selector can be implemented as an input selector of an Actel cell, with no additional Actel cells.

3.2.1 Mapping to the output selector

The output selector of the Actel cell is matched to the if-then-else triple at the node being mapped. The obvious mapping associates the if-branch with the control input, and the then- and else-branches with the two data inputs.

If the triple represents a two-input function, we have another choice—we can commute the triple before doing the mapping. Commuting can affect whether or not the OR-gate is usable, and can affect the polarity of the inputs. For example, $a + b'$ can be represented as (**if** a **then** TRUE **else** b') or (**if** b **then** a **else** TRUE).

We commute the triple if doing so will make it easier for us to use the OR-gate or the input selectors. The heuristics for deciding to use the commuted or uncommuted form of the triple are

- If one form of the triple already exists in either polarity, use it, and either create no cell or create an inverter, depending on the polarity needed.
- If only one of the forms has a free OR in the if-branch, use that form.
- Otherwise, if only one form has free selectors as the then- and else-branches, use that form.
- Otherwise, use the uncommuted form.

Having chosen the commute order for the triple, we now have to map each of the branches. The if-branch (a) maps to the OR-gate, and the then- and else-branches (b and c) map to the input selectors. For each branch, we can either choose to use the branch as an input to the Actel cell, making part of the cell be a simple buffer or inverter, or try to map the triple at the branch into the OR-gate or input selector.

3.2.2 Mapping to the OR-gate

The OR-gate output will be a or a'. We'll use the OR-gate as a simple buffer ($S_1 = 0$), if

- a already exists in some polarity (no need to use OR-gate), or
- all three branches of a are non-constant (can't represent a as an OR),
- a is not a free OR, and the node for a has three or more pointers to it in the DAG.

Otherwise, we express a as an OR or NOR, and connect the two inputs to S_0 and S_1.

An experiment was tried in which $a =$ (**if** x **then** y **else** z) was split into $xy + x'z$ to take full advantage of the OR-gate, whenever a had low fan-out, but this splitting increased the number of cells needed by about 0.5%.

If we choose to use only one input of the OR-gate, we can make its input either a or a'. The polarity choice here makes no difference to the cost of the cell we are currently mapping, but may affect how many cells are needed to generate the signal for S_0. We will have similar polarity choices to make for the control inputs of the input selectors S_a and S_b.

Currently, Amap uses the same simple heuristic as Xmap for making the polarity choice for a node:

- Use literals (primary inputs) in uninverted form.
- If either polarity is already implemented as an Actel cell output, use the one that exists.
- Otherwise, use the polarity chosen initially.

3.2.3 Mapping to the input selectors

After we have decided how to use the OR-gate, we are ready to map the then- and else-branches to the input selectors.

As with the mapping of the output selector, we have a choice of commute order for triples that represent two-input functions. The heuristics are fairly simple:

- If one form is a free selector and the other isn't, use the free selector.
- Otherwise, if only one form has an if-branch that exists in some polarity, use that form.
- Otherwise, use the uncommuted form.

We have a choice for each input selector: use it as a buffer for a single signal, or grab the next lower triple in the if-then-else DAG. A simple heuristic decides when to use the input selector as a buffer—use it as an inverting or non-inverting buffer, if

- either polarity of the node is already implemented,
- the node is not a free selector and has a high fan-out (≥ 3), or
- the polarities of the node's then- and else-branches would both be wrong.

Note that the last condition can never hold for a triple representing a two-input function.

3.2.4 XOR fixup and recursion

After choosing whether to use the input selectors as buffers, we check to see if the triple for the output selector is an XOR (that is, if the then-branch is the negation of the else-branch) and either of the two input selectors is a buffer. If one of the selectors is a buffer, we can get some savings by making the other selector a buffer also, and using the same signal for both S_a and S_b.

Next, we check each input node to see if it is a two-input function and the function with the two inputs commuted already exists. If so, we use the existing version, rather than creating a new one.

Finally, we map the inputs that aren't already implemented.

4 The XAmap algorithm

Of the 256 possible three-input logic functions, 5 take no cells, 208 can be done in one Actel cell, and the other 43 require two. An experiment was performed in which each of the 256 functions was minimized using the Printform transformations [4], then mapped by Amap. Amap got the 5 no-cell functions right, but mapped only 177 functions in one cell, and took two for the remaining 74.

Because Actel cells can implement almost any three-input function, and because Xmap with $f = 3$ ends up with about as many logic blocks as Actel cells from Amap [5], it is tempting to try to use Xmap to map to Actel cells. This is exactly what the XAmap algorithm does.

First, Xmap is called to generate the three-input functions (no merging is done), then each function is replaced by the corresponding Actel cell using a simple table lookup. If a logic block is created that doesn't correspond to an Actel cell, then one input has to be made available in both polarities, and the Actel cell programmed as a generic 4-to-1 selector with the other two inputs controlling the selector. If none of the inputs are currently available in both polarities, then an Actel cell programmed as an inverter needs to be added for one of them. XAmap choses to make the input with the highest fan-out double-rail, as it is most likely to be usable again.

One improvement to XAmap, which hasn't been implemented yet, would use a table lookup for cells that need double-rail input, rather than building the generic 4-to-1 selector. The table-lookup would minimize the number of connections to the cells that have a double-rail input. There would be no savings in the number of cells or routing complexity, but there could be a reduction in delay, due to reduced fan-out.

The table of Actel cell implementations for the XAmap algorithm was generated by a program that computed the function of each of the 5^8 ways of connecting 0, 1, a, b, or c to the inputs of an Actel cell. For each function, the wiring of the Actel cell that had the fewest inputs other than 0 or 1 was recorded. The table was set up in a way that favored implementations that used the output selector, rather than the input selectors, to reduce delay.

Note that the XAmap algorithm will always take at least as many cells as Xmap with $f = 3$, and will never have more than three different inputs to an Actel cell (four, if you count the double-rail signal as two). Although this seems to be throwing away much of the flexibility of the Actel cells, XAmap appears to work almost as well as Amap.

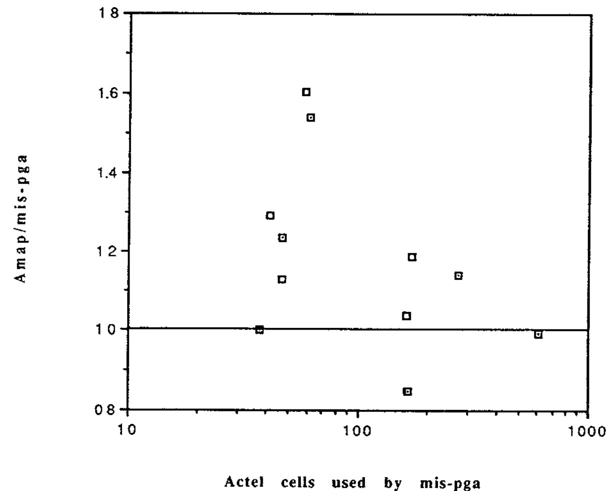

Figure 5.1: The number of Actel cells needed by Amap divided by the number needed by mis-pga for the benchmarks reported in [8].

An Actel cell can implement only 4,502 out of the 65,536 possible four-input functions—slightly less than 7% of them. Even if all inputs are provided double-rail, there are still 42,362 four-input functions that can't be implemented in one Actel cell. Because 64% of the four-input functions are unavailable, it is unlikely that changing XAmap from $f = 3$ to $f = 4$ will offer any advantages.

5 Results of benchmarks

Thirty-four benchmark circuits, including all the benchmarks reported for mis-pga [8], were mapped after minimizing. with misII's standard script. The results are tabulated in a technical report [5]. Figure 5.1 compares Amap with mis-pga for Actel cells [8]. The ratio of sizes is plotted, because the range is large enough to obscure the differences in a straight Amap vs. mis-pga scatter diagram.

The average performance of Amap is reasonable: 6% fewer cells than misII's mapper [2,8, Table 2], and about 8% more than mis-pga. Somewhat surprisingly, XAmap generates only 4% more cells than Amap (7% more on the benchmarks reported for mis-pga), and only 16% more cells than mis-pga, despite restricting itself to three-input functions.

Amap and XAmap are significantly faster than mis or mis-pga—about 25 and 19 times faster than misII and 586 and 445 times faster than mis-pga with Heuristic 3 and iterative improvement. It might be fairer to compare Amap and XAmap with Heuristic 2 of mis-pga without iterative improvement, which is mis-pga's fastest heuristic. Amap and XAmap do 20% and 14% better than Heuristic 2 without iterative improvement, but take 2–3 times the CPU time. The time ratio is really closer to 1–1.5, as the mis-pga results are from a 6 MIPS machine, and the new mappers were run on a 3 MIPS machine.

6 Choosing the cell type

In this section we examine the tradeoff between cell complexity and the number of cells needed for different selector-based cells. To evaluate a cell type, we need to combine the

cell type	grids/ cell	cells needed	total grids	grids* lg cells	3-input fcns
super-Actel	10	6484	64840	821055	236
Actel	9	6789	61101	777761	213
no-Or Actel	8	7137	57096	730898	197
2-to-1 selector	4	11711	46844	633130	32
2-to-1 selector with inverters	4	10594	42376	566613	54

Table 6.1: Area estimates for gate arrays based on different cell types. The final column is the number of distinct three-input functions that can be implemented with only one cell.

number of cells used by a mapping algorithm with an estimate of the area needed for each cell and an estimate of the area needed for wiring. For the cell types we'll consider, the cells themselves are small, and the area needed for them can be estimated by the number of *grids* (inputs and outputs) for the cell.

The routing area needed is proportional to the total number of grids times the average height of the routing channels. Unfortunately, average channel height is difficult to estimate before placement. Cell types that take more cells for a given circuit will require more channel height, but the height doesn't increase linearly with the number of cells, as the additional wiring will be mainly short wires. Lacking a good model for channel height, the analysis here will use lg(cells).

We can evaluate different cell types by making variants of the Amap algorithm to map to them, and mapping the same benchmarks for each type:

Actel The cell presented in Figure 2.1.

super-Actel Replace the OR-gate with a 2-to-1 selector, making the cell a full two levels of an if-then-else DAG.

no-OR Actel Omit the OR-gate from the Actel cell to get a four-input selector with three control inputs.

2-to-1 selector Use simple 2-to-1 selectors. Simple latches can still be done in one or two cells, but clear and preset functions would increase the size of the latch.

2-to-1 selector with inverters Use 2-to-1 selectors with optional inverters for the two data inputs. Such a cell can implement the if-then-else DAG directly. Programming the optional inverters may take a couple of extra switches or anti-fuses per input.

Table 6.1 summarizes the tradeoffs for the cell types described above.

7 Future Work

The high speed of the mappers makes them attractive for evaluating high-level minimization techniques. Circuit rearrangement can be done for area or delay reduction, and the entire circuit remapped to evaluate the changes—like the iterative improvement done in mis-pga. The techniques used in XAmap (ignoring the structure of a function, and looking only at the number of wires needed to encode the information used for computing it) may also be valuable in higher-level logic and state-machine minimization algorithms.

A more detailed study of programmable gate array architectures should be made, using better estimates for the cost of routing. Other selector-based cells, such as 4-to-1 and 8-to-1 multiplexers, should also be examined.

Another mapping approach that might be worth trying would be to use a bottom-up marking algorithm, like Xmap, but with overflow nodes defined by whether the function could be done with one Actel cell. This approach would combine the strengths of Amap and XAmap, but could be more difficult to program.

The results in this paper count only the number of cells used, without estimating routability or delay. Optimizing for cost measures that estimate these parameters would increase the value of the technology mappers. Unfortunately, the delays in programmable gate arrays are heavily dependent on the routing, because of the high resistance of the routing switches or anti-fuses. It will be difficult to find meaningful delay estimates before routing is done, and doing good placement and routing is too expensive to put in the inner loop of a logic minimizer.

Another direction for research is to develop mappers for sequential logic, generating complete mappings including feedback to make selectors act as registers.

Acknowledgements

Søren Søe read drafts of this article and provided useful comments.

References

[1] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. MIS: a multiple-level logic optimization system. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, CAD-6(6):1062–1081, Nov. 1987.

[2] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Technology mapping in MIS. In *ICCAD-87*, pages 116–119. IEEE Computer Society Press, Nov. 1987.

[3] K. A. El-Ayat, A. El Gamal, R. Guo, J. Chang, R. K. H. Mak, F. Chiu, E. Z. Hamdy, J. McCollum, and A. Hohsen. A CMOS electrically configurable gate array. *IEEE Jour. of Solid-state Circuits*, 24(3):752–762, June 1989.

[4] K. Karplus. Using if-then-else DAGs for multi-level logic minimization. In *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 101–118, March 1989.

[5] K. Karplus. Using if-then-else DAGs to do technology mapping for field-programmable gate arrays. Technical Report UCSC-CRL-90-43, Computer Engineering, Univ. of Calif., Santa Cruz, Sept. 1990.

[6] K. Karplus. Canonical forms of if-then-else dags are robustly path-delay-fault testable. In *International Workshop on Logic Synthesis*, Research Triangle Park, North Carolina, May 1991. submitted.

[7] K. Karplus. Xmap: a technology mapper for table-lookup field-programmable gate arrays. In 28th *Design Automation Conf.*, San Francisco, CA, June 1991.

[8] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic synthesis for programmable gate arrays. In 27th *Design Automation Conf.*, pages 620–625, Orlando, FL, June 1990.