Using if-then-else DAGS to do Technology Mapping for Field-Programmable Gate Arrays

Kevin Karplus*

UCSC-CRL-90-43
11 September 1990

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

ABSTRACT

This paper presents two new algorithms for doing mapping from multi-level logic to field-programmable gate arrays. One algorithm, Xmap, is for mapping to table-lookup gates (for example, the Xilinx chip); the other, Amap, is for mapping to selector-based architectures (for example, the Actel chip). Mapping to the Actel architecture can also be achieved by mapping to 3-input tables, and replacing them with equivalent Actel cells (XAmap).

The algorithms are based on an if-then-else DAG representation of the functions. The technology mappers differ from previous mappers in that the circuit is not decomposed into fan-out-free trees.

The gate counts and CPU time are compared with three previous mappers for these architectures: misII, Chortle, and mis-pga. The Xmap algorithm for table-lookup architectures gets 7% fewer cells than Chortle, 11% fewer than misII, and 14% fewer than mis-pga, and is 4.5 times faster than Chortle, 17 times faster than misII, and at least 150 times faster than mis-pga. The Amap algorithm for Actel cells use 6% fewer cells than misII and 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.

Keywords: Xilinx, Actel, logic minimization, Xmap, Amap, XAmap, if-then-else DAG, table-lookup gates

---

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

Contents

1 Why a new technology mapper?

Previous generations of logic minimization tools have generally used technology mappers that work by choosing out of a library of available cells [GBdH86,Keu87,DGR+87,BJ88]. Field-programmable gate arrays, such as those made by Xilinx and Actel, do not use a library of different cell types, but use an array of identical cells, each of which can be used quite flexibly. The cell-library-based mappers do not work particularly well when mapping to such flexible cells, and so dummy cell libraries are usually created, where each library entry is one of the many ways of using the basic cells of the gate array. The cell-library approach allows existing technology mappers to be used, but does not scale well as the size of the basic cells increases, because the library tends to grow exponentially with the size of the basic cell.

At the 1990 Design Automation Conference, two papers were presented on new technology mappers for programmable gate arrays: Chortle and mis-pga [FRC90,MNS+90]. These techniques avoid the need for a cell library, but are still based on decomposing the circuit into fan-out-free trees before mapping. That is, the DAG is decomposed into a number of trees, each of which has only one output. Any node which has fan-out greater than one in the original DAG will be the root of one of these trees. Even if the trees are optimally mapped, the decomposition may be worse than a non-optimal mapping of the original DAG.

Because the problem is relatively new, there are many different approaches being tried, and considerable improvement is still possible. The approach taken in this paper is to build quick-and-dirty technology mappers that get good results without rearranging the circuit. The intent is to have a technology mapper that is fast enough to be included in an iterative improvement loop for a higher-level logic minimization algorithm.

Two new algorithms are presented in this paper: one for mapping to table-lookup-based gate arrays, such as Xilinx, and one for mapping to selector-based gate arrays, such as Actel. Both algorithms handle only combinational logic.

The Xmap algorithm is much faster than Chortle or mis-pga, and the results are better. The results on Actel arrays are mixed—Amap and XAmap are very much faster, but not quite as effective as mis-pga.

2 Conversion from BLIF to If-then-else DAGs

The data structure used to represent the circuits is a multiply-rooted if-then-else directed acyclic graph [Kar89]. Each root corresponds to a primary output of the circuit, and each leaf to a primary input. Intermediate nodes can be thought of as 2-to-1 selectors, with the control input connected to the node on the if-branch, and the "1" and "0" inputs connected to the then- and else-branches. The connections between the selectors can be either wires or inverters—the polarity of a function is stored as a label on the edge pointing to the DAG node, rather than using extra nodes for inversion.

In order to compare the new mapping algorithms fairly with existing mappers, they have been run on the output of the misII logic minimizer [BRSW87], which is in Berkeley Logic Interchange Format (BLIF). Because BLIF represents a circuit as a network of sum-of-products representations, rather than as an if-then-else DAG, a conversion needs to be made.

Building an if-then-else DAG from a network of gates is easy if each gate is described as an if-then-else DAG—we simply glue together the gate functions to build the DAG for the entire circuit. The only tricky part is converting the sum-of-products descriptions of the gates into if-then-else DAGs. We have several choices:

- Build a canonical DAG for the function expressed by the gate. For gates of the form $ab + cd + ef + gh + \cdots$, the wrong variable ordering can cause an exponential blowup in size.
- Preserve the original and-or structure of the sum-of-products expression. This is guaranteed not to be exponentially large, but is much larger than it needs to be.
- Build a partially factored expression for the gate.

The conversion technique used was described at the 1989 Caltech conference [Kar89]. The terms of the sum-of-products expression are collected in a set T. Then the variables are sorted in decreasing order of the frequency with which they occur in the terms. Next, a recursive function is applied to T to get an if-then-else DAG E. The terms of T are sorted, grouping together those that don't use the first input variable (T_d), those that use v_1' (T_0), and those that use v_1 (T_1). We then strip the first variable off the terms in each group, and apply the routine recursively to get expressions E_d, E_0, and E_1. We build the expression E as (**if** E_d **then** TRUE **else** (**if** v **then** E_1 **else** E_0)).

This algorithm is similar to the popular method of factoring out one-literal cubes, and produces expressions that are often significantly smaller than either the canonical form or the straight sum-of-products form. A similar technique is used to create binary decision diagrams in mis-pga, but binary decision diagrams cannot represent the separation of E_d from E_0 and E_1, and so much of the factorization is lost.

One major difference between the new mappers and Chortle or mis-pga is that the new mappers make essentially no changes to the DAG that represents the circuit to be implemented. The logic blocks found by the new mappers are possibly overlapping sub-DAGs of the DAG for the entire circuit. This fidelity to the original circuit representation is not inherently a virtue, but it does make the mapping very quick. It also makes the mappers somewhat sensitive to the way in which the conversion to a DAG is done.

3 Mapping to table-lookup logic blocks

One popular approach for programmable gate arrays is to provide logic blocks that can implement any Boolean function of up to f variables. The logic blocks are easily implemented as a 2^f-bit ROM or RAM for the truth table and a selector controlled by the f inputs. Typically, the fan-in limit f is around 4 or 5.

Larger blocks take significantly more space for the truth tables, but fewer of them are needed, and expensive routing resources are saved. Based on the benchmark results for both Xmap and Chortle, it appears that 15–18% fewer blocks are needed if they are 5-input blocks rather than 4-input blocks. A technique for deciding on a good block size is presented in Section 6.2.

Because blocks are often used with fewer than the maximum number of inputs, Xilinx allows their blocks to be split into two $(f-1)$-input functions, as long as the total number of inputs to the block is still no more than f. The truth table storage space remains the same, and the selector circuitry is only slightly more complex. Early versions of the Xilinx chip used blocks with four inputs that could be split into two three-input functions. More recent chips use five-input blocks that can be split into two four-input functions.

The algorithm for mapping to logic blocks works in three passes. On the first pass, some nodes are marked as being outputs from logic blocks, and a set of f or fewer marked nodes is stored as gate inputs for each node. Note that for any function g, both g and $\neg g$ are represented by the same node, and so no polarity choices have been made yet. On the second pass, polarities are chosen for each marked node, and the function of its gate is determined—the stored set of gate inputs may be used, or a new set chosen if a smaller one is found. The final pass looks for functions that don't use all f inputs and tries to share logic blocks.

3.1 Xmap algorithm—Marking pass

The marking pass can be thought of as a bottom-up lazy marker. A traversal of the DAG is done, making sure that nodes are visited only after all their inputs have been visited. Nodes that are principal inputs or outputs of the circuit are always marked, but other nodes are not unless forced to by some ancestor.

For each node we keep track of the fan-in set: a set of marked nodes used to represent the function of that node. If the node is a primary input, then the set contains the node itself, otherwise it contains the signals we would need to use as inputs to a logic block that could compute the function. When we decide to mark a node, we save the fan-in set as the gate inputs, and change the fan-in set to be the singleton containing just the node. Note that the marking phase pays no attention to what the function is—only to the variables needed to compute it.

The fan-in set for a node can be computed as the union of the fan-in sets of the children of the node, as long as that union is no more than the fan-in limit for logic blocks. Let's call a node whose fan-in set is larger than the allowable fan-in an *overflow node*.

When we reach an overflow node in the postorder traversal, we have to reduce the fan-in by marking one or more of its descendants, thus hiding the nodes below that descendant. Note that we only have to examine descendants down to the nodes in the fan-in set—a fairly small piece of the DAG.

The fan-in reduction for overflow nodes is done in two stages. The first stage attempts to reduce the fan-in by marking only nodes with high fan-out in the if-then-else DAG. This heuristic is called repeatedly until the fan-in is reduced to f or less, or no progress is made. If the fan-in limit is still not met, the children of the node are marked one at a time until the fan-in limit is met. Because each if-then-else triple has at most three children, eventually the fan-in limit will be met for $f \geq 3$.

To extend the algorithm to $f = 2$, we first preprocess the entire DAG, replacing all three-input if-then-else triples (**if** a **then** b **else** c) with either $ab + a'c$ or $(a + c)(a' + b)$. This guarantees that each node has at most two children, not counting TRUE and FALSE, and so marking will eventually reduce the fan-in to two.

Conversion to two-input gates could be done before calling Xmap for higher fan-in blocks as well. Doing so has mixed results: the number of blocks before merging reduces by about 0.3%, but the number of blocks after merging increases by about 2.3%. The initial reduction probably results from some additional sharing of the new triples generated. The results in Section 5 use the conversion to two-input gates only for $f = 2$.

The quality of the mapping algorithm depends heavily on the details of the heuristics used to choose which nodes to mark.

For the heuristic that marks large fan-out nodes, we use a recursive procedure, reduce_fan-in(n, h), applied to the node n.

1. We mark all children of n that have a large fan-out (fan-out $\geq h$). A new set of block outputs for n is computed, and recorded if it is smaller than the old set. We return success if the fan-in of n has been reduced.

2. If marking the high-fan-out children doesn't reduce the fan-in of the node, we recursively apply reduce_fan-in$(child(n), h + 1)$ until one of them succeeds, then recompute the fan-in of n and return success if its fan-in set is also reduced. Note that the effects of marking a node are not propagated to all the nodes that might be affected, but only to the immediate parent that requested the fan-in reduction.

The reduce_fan-in(n, h) procedure is repeated for the overflow node until the fan-in is sufficiently reduced or the procedure fails to to reduce the fan-in. The criterion for high-fan-out, h, seems to work well if it is set at 2 for the first level, and incremented by one for each deeper layer of recursion. It also works well to use a constant $h = 3$, and not increase it in the recursion, but the benchmarks in Section 5 were done with an increasing limit.

The marking algorithm is fast, as each node is visited exactly once by the outer traversal, and the fan-in reduction algorithm only visits a few nodes.

3.2 Xmap algorithm—Building the logic blocks

After nodes have been marked, another traversal is done to choose polarities and assign gate functions. The polarities of the outputs are already set, and the polarities of the inputs are all positive. For all the intermediate nodes the choice of polarity is arbitrary, as long as it is consistent in all uses. The only times that inverters are created are when a primary output is the negation of a primary input or both polarities of a primary output are needed.

For each marked node that isn't a primary input we need to create a logic block, choose appropriate inputs for it, and determine its function. A legal set of inputs for a logic block form a vertex cut in the DAG separating the node from the primary inputs. The cut must contain no more than f nodes, and ideally, we would like it to be as small as possible, so that the function is more likely to be merged with another into a shared logic block.

We know that a legal vertex cut exists—the fan-in set that was recorded when the node was marked is such a cut. Doing a full min-cut algorithm to find the smallest legal input set for each logic block seems overly complex, but there is another vertex set that is easy to generate: the cut closest to the marked node. A simple depth-first traversal stopping at marked nodes will find this cut quickly. Whichever of the two cuts (the fan-in set or the closest cut) is smaller is the one that is used as the inputs for the gate.

If we always used the closest cut, then each logic block would cover some portion of the DAG with no overlap between blocks, minimizing the amount of duplicated logic. Unfortunately, the closest vertex cut is not guaranteed to meet the fan-in limit.

Once we have chosen the inputs for a logic block, the if-then-else DAG for its function is simply the DAG for the node truncated at the set of inputs, with appropriate inversions of pointers at the inputs and outputs to match the assigned polarities.

3.3 Xmap algorithm—Sharing logic blocks

The Xilinx architecture allows two functions to share a single combinational logic block (CLB) if each function uses no more than $f-1$ inputs, and the combined functions have no more than f total inputs. The mis-pga mapper takes advantage of this feature.

Murgai *et al.* correctly identified the problem as finding a maximum matching between mergeable blocks [MNS+90], defining two blocks as being *mergeable* if

- each has at most $f-1$ inputs,
- the number of common inputs is $f-2$, and
- the total number of inputs is f.

The second requirement is an unnecessary restriction, as two four-input functions can share a five-input CLB if they have identical inputs. Of course, to handle two $(f-1)$-input functions one of the inputs will have to be routed to two of the CLB inputs.

The matching algorithm used with Xmap is not a true maximum matching algorithm, as such an algorithm could take $n^{2.5}$ steps to do the matching for a network with n logic blocks, and the maximal matching algorithms are fairly complicated. Instead, Xmap uses a quick-and-dirty greedy algorithm to get a good, but usually not maximal, matching.

The algorithm takes a set of logic blocks L and two parameters: i, the maximum number of inputs for each function in a shared CLB, and t, the maximum total number of inputs in a shared CLB. First, all blocks with more than i inputs are removed from L as being unmergeable. The remaining set is sorted by the number of inputs to each block.

The block with the most inputs is removed from L and mergeability is checked with all remaining blocks, in decreasing order of number of inputs. If a legal merging is found, then the other block is also removed from L. The removing of blocks is continued until L is empty.

Trying to merge blocks with high fan-in first serves two purposes: it helps ensure that easy-to-match blocks (blocks with few inputs) are kept available for later matching, and it reduces routing demands by increasing the number of block inputs that are shared.

Theoretically, the mergeability check could be done as many as $n(n-1)/2$ times, if all n blocks have few enough inputs to be pairable but no matches are actually possible. In practice, n is small enough and the inner loop fast enough, that the matching takes insignificant time.

4 Mapping to Actel cells

4.1 The Actel cell

One approach to designing field-programmable gate arrays is to use many copies of a small, versatile cell. This approach requires many connection points, and so is most appropriate when the cost of connections is low. The Actel chip is designed using this approach [EAGG+89]. Their basic cell (illustrated in Figure 4.1) uses selectors as the basic gates, and can be configured for either combinational logic or storage. We will only consider the uses for 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 [EAGG+89, page 753]. All the two input functions are trivially implementable, but only 213 of the 256 different three-input functions can be implemented in a single cell. The missing functions are shown in 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.

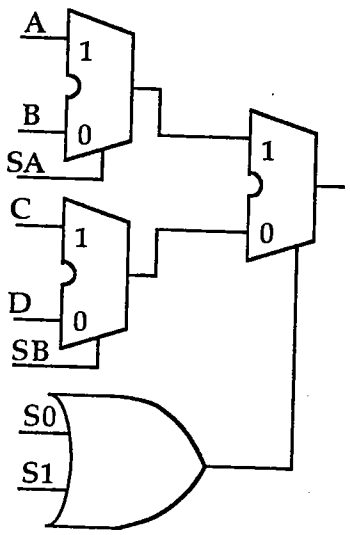

Figure 4.1: Basic cell used in Actel chips field-programmable-gate-array chips.

function	symmetries
$abc + a'b'c'$	1
$(abc + a'b'c')'$	1
$a \oplus b \oplus c$	1
$(a \oplus b \oplus c)'$	1
$abc + (a \oplus b \oplus c)'$	1
$(a \oplus b \oplus c)(abc)'$	1
$a'b'c' + (a \oplus b \oplus c)$	1
$(a \oplus b \oplus c)'(a'b'c')'$	1
$ab \oplus c$	3
$(ab \oplus c)'$	3
$a'b'c + (a \oplus b \oplus c)'$	3
$(a \oplus b \oplus c)(ab'c')'$	3
$a'bc + (a \oplus b \oplus c)$	3
$(a \oplus b \oplus c)'(abc')'$	3
$(abc)'$	1
$a'b' + b'c' + a'c'$	1
$a'(b \oplus c)$	3
$a'(b' + c')$	3
$a'bc + (b \oplus c)$	3
$a' + (b \oplus c)$	3
$a' + (b \oplus c)'$	3

Table 4.1: The 43 three-input functions that cannot be implemented with a single Actel gate. The first 26 can't be implemented in either polarity; the negations of the remaining 17 can be implemented in one gate.

4.2 Amap algorithm—overview

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 actual mapping in the second pass work a little better.

The second pass is a top-down recursive function that is called once for each output. The function is passed a node of the DAG, and generates an Actel cell whose output corresponds to that node. The function matches the Actel cell to some portion of the top of the DAG, making the inputs to the cell correspond to lower nodes in the DAG, then calls itself on the inputs that haven't already been mapped to Actel cells. Most of the complexity of the function comes in deciding how much of the DAG to cover.

4.3 Amap algorithm—local manipulation

The mapper does a traversal of the DAG, looking for triples that have only two non-constant children. These triples represent two-input commutative functions. Each such triple is checked to see if exactly one of its inputs is a literal. If so, the triple is commuted, if necessary, to make the literal be in the if-part, rather than in the then- or else-parts of the triple. 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. The commute order determined in this pre-pass 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.

Because the symbol table that stores the nodes will merge identical triples, but not triples that differ only in the commute order, standardizing the commute order may reduce the size of the DAG slightly. This reduction is not enough to be responsible for the improved mapping, as the number of nodes was reduced by only about 0.1%, noticeably less than the 0.5% improvement in the number of Actel cells. Doing the commute ordering before running Xmap results in a 0.1–0.2% improvement in mapping to Xilinx cells, about what one would expect from the change in the number of nodes. The same commute ordering was done for both Xmap and Amap in the results reported in Section 5.

Because standardizing the commute order when one input was a literal seemed to work well, some attempts were made to reduce the DAG further by standardizing the commute order when neither or both inputs were literals. These experiments resulted in 0.5–2% increases in the number of Actel cells produced by the mapper, and so were discarded.

After the commute order is chosen, a preferred polarity is chosen for each node in the DAG. The normalization of pointers done in storing the triples [Kar89, p. 105] ensures that all if- and then-branches are non-inverting pointers, and so every node has a non-inverting path through then-branches to a principal input or to 1. The preferred polarity for each node is to use it uninverted, except for the outputs, which are used in whatever polarity is needed.

Various heuristics were tried to find a better initial assignment of polarities, but none of them decreased the number of Actel cells by more than about 0.1%. Doing a random assignment of initial polarities costs only about 0.5%, and so the initial assignment of polarities is apparently not particularly important. I found this surprising, as I expected the polarity assignment to be crucial.

The concepts of commute order and polarity are important, but the first pass has surprisingly 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 about polarity and commute order independent of the first pass.

4.4 Amap algorithm—matching Actel cell to DAG

After the commute order and initial polarity assignment 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 either case, Amap either creates no cells, or creates a cell set up 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 only ever use one input to the OR-gate, we'll need about 5% more cells than if we use both inputs wisely. Sections 4.5, 4.6, and 4.7 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 triple 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.

4.5 Amap—mapping to the output selector

The top 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 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 maps to the OR-gate, and the then- and else-branches 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.

4.6 Amap—mapping to the OR-gate

Let's call the if-branch a. We will use a or its complement as an input to the Actel cell ($S_0 = a$ or $S_0 = a'$ and $S_1 = 0$), 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 (don't expand high fan-out nodes).

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

An attempt was made to split a into $xy + x'z$ to take full advantage of the OR-gate, whenever it was a three-input if-then-else triple with low fan-out, but this 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.

Once a polarity decision has been made for a node, it is recorded to maintain consistency.

4.7 Amap—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 top 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 either 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 is used to decide 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, because the polarity of a constant 0 or 1 is never wrong, and the then- and else- pointers from an XOR triple point to the same node but with differing polarities, one of which must be right.

4.8 Amap—XOR fixup and recursion

After choosing whether to use the input selectors as buffers, we check to see if the top triple is an XOR (then-branch 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 will be generating an Actel cell corresponding to some polarity of that XOR input, so we might as well make the other selector a buffer also, and use the same signal for both S_a and S_b. This XOR fixup rarely applies, and so could be omitted with almost no loss in average performance.

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.

4.9 The XAmap algorithm

We know that of the 256 3-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 [Kar89], 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, 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 with a simple table lookup. If a logic block is created that doesn't correspond to an Actel cell, then one of the inputs 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 slight 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. In this way, we could minimize the number of connections to the cells that have a double-rail input. There would not be any 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.

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

Tables 7.1 and 7.2 compare the Xmap, Amap, and XAmap algorithms with results for Chortle [FRC90] and mis-pga [MNS+90,Mur90]. In all cases the benchmark circuits were minimized with misII's standard script before mapping. The time ratios given for mis-pga should probably be doubled, as the programs were run on different machines—Xmap on a SUN 3/80 (a 3 MIPS machine), and mis-pga on a Vax 8800 (a 6 MIPS machine). Chortle and misII were run on a SUN 3/60, and so the CPU times should be directly comparable with Xmap.

5.1 Xmap performance

The Xmap algorithm consistently outperforms Chortle, even without merging blocks, averaging 7% fewer blocks. It is interesting to note that Xmap outperforms Chortle even for $f = 2$, where the Xmap algorithm does almost nothing after the initial removal of 3-input if-then-else triples.

The Xmap algorithm gets the same number of blocks as mis-pga before merging, but gets 14% fewer after merging. Xmap reduces the number of logic blocks by over 30% with merging, and mis-pga only achieves about 21% reduction. One might think that Xmap's merging algorithm is better than mis-pga's, but this is unlikely, as mis-pga uses a maximum matcher, and Xmap only a heuristic approximation to one. The difference is probably that Xmap makes more of an effort to minimize the fan-in of gates, and so more gates are mergeable.

The Xmap algorithm is significantly faster than Chortle or mis-pga. Overall, Xmap is about 4.5 times faster than Chortle, but Xmap does not slow down as the fan-in limit f increases, while Chortle's time appears to increase linearly with f. On the most interesting cases ($f = 5$), Xmap is about 7 times faster than Chortle. Compared to mis-pga, Xmap is about 150 times faster, and manages to complete the merging on all benchmarks, while mis-pga runs out of memory despite a generous swap space.

Francis *et al.* also reported times for using misII to map to a large library of cells that simulated mapping to arbitrary functions [FRC90, Tables 1–4]. For the benchmarks they reported on, Xmap is about 17 times faster than misII, and gets 11% fewer cells.

5.2 Amap and XAmap performance

The average performance of Amap is reasonable: 6% fewer cells then misII's mapper [MNS+90, Table 2], and about 8% more than mis-pga. Somewhat surprisingly, XAmap generates only 4% more cells than Amap, and only 16% more cells than mis-pga, despite restricting itself to three-input functions.

The Amap and XAmap algorithms 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 have been fairer to compare Amap and XAmap with Heuristic 3 of mis-pga *without* iterative improvement, as the Amap and XAmap algorithms do no re-minimization of the circuit. The Amap and XAmap algorithms do 12% and 6% better than Heuristic 3 without iterative improvement, and are about 73 and 55 times as fast.

6 Hints for choosing or designing programmable gate arrays

This section evaluates different programmable gate array architectures, examining the tradeoff between cell complexity and the number of cells needed.

The mapping algorithms give us estimates of the number of cells needed for a variety of problems, and are easily modified for slightly different architectures. To evaluate an architecture, we need to combine these estimates with an estimate of the area needed for each cell and an estimate of the area needed for wiring.

6.1 Designing a selector-based architecture

For selector architectures, the cells themselves are quite small, and the area needed for them is can be estimated by the number of *grids* (number of inputs plus number of outputs) needed for the cell. The total routing area needed is proportional to the total number of grids times the height of the channel.

Unfortunately, the necessary height for the channel is difficult to estimate. For a given circuit, it will increase somewhat for architectures that take more cells, but probably not 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 just the number of grids as the estimate for area.

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

- Using the Actel cell, Amap needs 6789 cells to implement all 34 benchmarks. With eight inputs and one output per cell, 61,101 grids are needed. Actel cells can implement 213 of the 256 three-input functions.
- Replace the OR-gate with 2-to-1 selector, making the cell a full two levels of an if-then-else DAG. The 34 benchmarks need 6484 of these super-Actel cells, 3% fewer than with the Actel cells. The number of grids increases from nine to ten per cell, increasing the total number of grids by 6%. Super-Actel cells can implement 236 of the three-input functions in one cell.
- Omit the OR-gate from the Actel cell to get a 4-input selector with three control inputs. Mapping all 34 benchmarks takes 7137 cells, which is 5% more than with Actel cells. This reduced Actel cell uses eight grids instead of nine, reducing the total number of grids by 7%. Reduced Actel cells can implement 197 of the three-input functions in one cell, but can implement all of them if one input is provided double-rail, and so the XAmap algorithm could be applied.

cell type	grids/cell	cells needed	total grids
super-Actel	10	6484	64840
Actel	9	6789	61101
no-Or Actel	8	7137	57096
2-to-1 selector	4	11711	46844
2-to-1 with inverters	4	10594	42376

Table 6.1: Area estimates for gate arrays based on five different cell types. The second column is the number of inputs plus outputs for each cell, the third column is the number of cells needed for all 34 benchmarks by an appropriate variant of the Amap algorithm, and the fourth column is the estimate of area needed for the circuits.

- Use simple 2-to-1 selectors. Mapping all 34 benchmarks takes 11711 selectors—72% more cells than using Actel cells. With only four grids per cell, the total number of grids is reduced by 23%. Simple latches can still be done in one or two cells, but clear and preset functions would increase the size of the latch. Selectors can implement only 32 of the 256 three-input functions, making XAmap useless.
- Use 2-to-1 selectors with optional inverters for the two data inputs. Such a cell can implement the if-then-else DAG directly. The number of if-then-else triples in the 34 benchmarks is 10594, 56% more cells than using Actel cells and 10% fewer than using selectors without input inverters. The number of grids is 31% smaller than with Actel cells. Programming the optional inverters may take a couple of extra switches or anti-fuses per input. Selectors with optional input inverters can implement 54 of the three-input functions.

Table 6.1 summarizes the tradeoffs for the architectures described above. Based on the number of grids, the 2-to-1 selectors with programmable input inverters seem to be the best cells. Even with reasonable increases in channel height, using these cells should still take significantly less area than using Actel cells.

6.2 Choosing lookup table size

The Xmap algorithm gives us some guidance in designing new chips using a table-lookup array architecture. For each additional input to a logic block, we double the size of the truth-table memory and increase the number of wires that need to be routed to the block, but decrease the number of blocks needed to implement the function. If we assume that the cost of the wiring is proportional to the number of inputs and outputs of the block, then the cost of using a single-output, f-input block is $c_b 2^f + c_y(f+1)$. Blocks that have two outputs have cost $c_b 2^f + c_y(f+2)$. The constants c_b and c_y depend on the relative cost of bits of the truth table and of the wiring.

We can use this cost estimate to choose a good block size. The technique is illustrated in Table 6.2, which shows that 5-input blocks with merging are optimal for the arbitrary set of constants chosen. Of course, other constants and other benchmarks could give quite different results.

6.3 Comparing Actel and Xilinx

It is interesting to compare apples and oranges: what is the ratio of Actel cells to Xilinx logic blocks needed for a given function? Comparing Xmap and Amap on all 34 benchmarks, a five-input Xilinx logic block appears to be the equivalent of about 2.25 Actel cells for combinational logic. Because the Xilinx cells also contain two flip-flops, and one or two additional Actel cells are needed for each bit of storage, the actual ratio of the number of cells needed to implement a sequential function is somewhat higher. If we assume that every one of the 1558 principal outputs in the 34 benchmarks needs to be stored in a two-cell register, the ratio goes up to 3.3 Actel cells

block type	cost per block	blocks needed	total cost/10^6
2-input	124	12013	1.49
3-input	168	6961	1.17
3-input with merge	208	5983	1.24
4-input	216	5376	1.16
4-input with merge	256	4212	1.08
5-input	272	4411	1.20
5-input with merge	312	3027	0.94
6-input	344	3731	1.28
6-input with merge	384	2479	0.95

Table 6.2: Illustration of method for choosing a logic block size when designing a new gate-array chip. Constants were arbitrarily chosen as $c_b = 1$ and $c_y = 40$. The number of blocks needed is the sum of the numbers found by Xmap for all 34 benchmarks.

per five-input Xilinx block. Typical examples will probably fall somewhere between these extreme assumptions.

7 Future Work

The high speed of the mappers make them attractive for evaluation of high-level minimization techniques. Circuit rearrangement can be done for area or delay reduction, and the entire circuit remapped to evaluate the changes—the sort of iterative improvement done in mis-pga.. The techniques used in Xmap (ignoring the structure of a function, and looking only at the number of wires needed to encode the information needed to compute it) may also be valuable in high-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. The analysis in Section 6.1 suggests that a smaller cell would be superior, but the analysis may be underestimating the cost of routing the greater number of nets.

The mappers described in this paper are concerned only with the number of logic blocks used, not with routability or delay. Adding 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 come up with meaningful delay estimates before routing is done, and good placement and routing is probably too expensive to put in the inner loop of a logic minimizer.

Acknowledgements

I'd like to thank Martine Schlag for an afternoon discussion that crystallized some of the ideas in Xmap. Søren Søe read drafts of this report and provided useful comments.

References

[BJ88] M. R. C. M. Berkelaar and J. A. G. Jess. Technology mapping for standard-cell generators. In *IEEE International Conference on Computer-Aided Design ICCAD-88*, pages 470–473, Santa Clara, CA, 7–10 November 1988.

[BRSW87] Robert K. Brayton, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert R. Wang. MIS: a multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-6(6):1062–1081, November 1987.

[DGR+87] Ewald Detjens, Gary Gannot, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert Wang. Technology mapping in MIS. In *IEEE International Conference on Computer-Aided Design ICCAD-87*, pages 116–119, Santa Clara, CA, 9–12 November 1987. IEEE Computer Society Press.

[EAGG+89] Khaled A. El-Ayat, Abbas El Gamal, Richard Guo, John Chang, Ricky K. H. Mak, Frederick Chiu, Esmat Z. Hamdy, John McCollum, and Amr Hohsen. A CMOS electrically configurable gate array. *IEEE Journal of Solid-state Circuits*, 24(3):752–762, June 1989.

[FRC90] Robert J. Francis, Jonathan Rose, and Kevin Chung. Chortle: a technology mapping program for lookup table-based field programmable gate arrays. In *ACM IEEE* 27^{th} *Design Automation Conference Proceedings*, pages 613–619, Orlando, FL, 24–28 June 1990.

[GBdH86] David Gregory, Karen Bartlett, Aart de Geus, and Gary Hachtel. Socrates: a system for automatically synthesizing and optimizing combinational logic. In *ACM IEEE* 23^{rd} *Design Automation Conference Proceedings*, pages 79–85, Las Vegas, NV, 29 June–2 July 1986.

[Kar89] Kevin Karplus. Using if-then-else dags for multi-level logic minimization. In Charles L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 101–118, Pasadena, CA, 20-22 March 1989.

[Keu87] Kurt Keutzer. DAGON: Technology binding and local optimization by DAG matching. In *ACM IEEE* 24^{th} *Design Automation Conference Proceedings*, pages 341–347, Miami Beach, FL, 28 June–1 July 1987.

[MNS+90] Rajeev Murgai, Yoshihito Nishizaki, Narendra Shenoy, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Logic synthesis for programmable gate arrays. In *ACM IEEE* 27^{th} *Design Automation Conference Proceedings*, pages 620–625, Orlando, FL, 24–28 June 1990.

[Mur90] Rajeev Murgai. Personal communication. Electronic mail, August 1990.

name	$f=2$ Xmap	$f=2$ Chortle	$f=3$ Xmap -merge	$f=3$ Xmap +merge	$f=3$ Chortle	$f=4$ Xmap -merge	$f=4$ Xmap +merge	$f=4$ Chortle
10bitreg	20		20	10		20	10	
10count	85		54	41		47	31	
180degc	102		60	45		45	29	
3to8dmux	92		54	44		41	33	
4-16dec	27		26	15		23	14	
4cnt	54		33	28		23	19	
5xp1	105		58	49		46	34	
8bappreg	89		57	46		47	32	
8count	123		66	53		51	36	
9bcasc2	123		72	55		54	39	
9bcsac1	113		69	52		51	36	
9symml	193	199	108	93	112	79	65	78
9symml-good	52		23	22		19	14	
C1908	301		188	167		154	110	
C499	184		118	105		95	75	
C5315	1261		655	577		525	407	
alu2	358	382	209	177	218	154	120	159
alu4	627	691	362	311	405	264	202	286
apex6	647	665	370	320	390	266	222	261
apex7	179	200	108	96	126	80	59	94
arbiter	102		60	45		45	29	
bw	149		98	77		79	52	
clip	97		57	46		46	34	
count	95	113	55	47	56	39	28	49
des	2841	3049	1644	1418	1805	1278	1072	1225
duke2	294		184	156		144	105	
f51m	107		64	54		48	37	
frg1	107	111	61	58	60	45	38	43
frg2	689	740	411	350	452	305	243	333
k2	758	811	468	418	480	385	276	379
pair	1268	1441	706	614	851	543	446	635
rd84	170		98	85		75	56	
rot	526	578	306	272	357	233	182	261
vg2	75		39	37		27	27	
total(C)	8288	8980	4808	4174	5312	3671	2953	3803
total	12013		6961	5983		5376	4212	

Table 7.1: Number of gates used by different mappers for several benchmarks. Subset C is the set of benchmarks for which Chortle results are available.

name	Xmap -merge	Xmap +merge	$f = 5$ Chortle	mis-pga -merge	mis-pga +merge	Actel Amap	Actel XAmap	Actel mis-pga Heur. 3
10bitreg	20	10		21	10	20	20	
10count	36	23		33	23	52	54	
180degc	39	22		35	21	54	60	
3to8dmux	36	25		41	30	45	54	
4-16dec	24	12		20	12	23	26	
4cnt	20	13		21	17	32	33	
5xp1	34	25		31	23	53	58	41
8bappreg	39	27		37	27	58	57	
8count	42	26		35	20	69	68	
9bcasc2	45	29		52	29	69	72	
9bcsac1	42	29		50	34	62	69	
9symml	61	48	63			102	110	
9symml-good	12	11				24	23	
C1908	125	83				204	200	172
C499	84	61		66	50	141	139	166
C5315	404	294		497	(fail)	604	685	610
alu2	126	90	131	129	102	196	210	
alu4	223	150	238	235	229	336	365	
apex6	212	159	234	243	191	366	370	
apex7	70	45	73	64	50	112	109	
arbiter	39	22		35	21	54	60	
bw	56	37				94	102	61
clip	37	26				53	57	47
count	31	22	47	31	28	62	55	
des	1057	694	1075			1605	1652	
duke2	125	82		128	105	169	188	163
f51m	33	26				58	64	47
frg1	29	29	34	36	28	61	61	
frg2	291	183	278	246	192	406	412	
k2	333	218	335	349	289	482	473	
pair	417	307	504	430	(fail)	684	729	
rd84	55	43		40	32	93	101	58
rot	192	136	230	200	153	309	309	271
vg2	22	20		30	21	37	39	37
total(C)	3042	2081	3242			4721	4855	
total(X)	3030	2102		3135		4649	4876	
total(XM)	2209	1501		2208	1737	3361	3462	
total(A)	1167	833				1815	1942	1673
total	4411	3027				6789	7084	

Table 7.2: Number of gates used by different mappers for several benchmarks. Subset C is the set of benchmarks for which Chortle results are available. Subsets X, XM, and A are subsets for which mis-pga results are available for Xilinx cells before merging, Xilinx cells after merging, and Actel cells.

name	Xmap (including merge) $f=2$	$f=3$	$f=4$	$f=5$	Amap	XAmap	subsets
10bitreg	0.78	0.83	0.82	0.83	0.75	1.13	XT
10count	1.38	1.23	1.25	1.18	1.18	1.70	XT
180degc	1.68	1.40	1.35	1.35	1.23	1.80	XT
3to8dmux	1.52	1.37	1.38	1.32	1.17	1.78	XT
4-16dec	0.88	0.85	0.85	0.92	0.83	1.25	XT
4cnt	1.03	0.92	0.90	0.92	0.78	1.32	XT
5xp1	1.42	1.15	1.17	1.12	1.00	1.58	XT A
8bappreg	1.47	1.38	1.38	1.37	1.27	1.80	XT
8count	1.93	1.60	1.57	1.55	1.42	2.05	XT
9bcasc2	1.98	1.67	1.65	1.60	1.47	2.15	XT
9bcsac1	1.82	1.58	1.53	1.50	1.40	2.02	XT
9symml	2.35	1.97	1.93	1.92	1.65	2.50	C
9symml-good	0.77	0.60	0.60	0.62	0.48	0.95	
C1908	4.32	3.88	3.92	3.97	3.60	4.57	A
C499	2.93	2.65	2.72	2.82	2.52	3.17	XT A
C5315	17.73	13.75	14.60	13.42	11.68	15.37	XT A
alu2	4.60	3.90	3.85	3.83	3.27	4.62	C
alu4	7.97	6.80	6.63	6.70	5.72	7.82	C
apex6	9.22	7.85	7.78	7.63	6.87	8.90	C XT
apex7	2.63	2.33	2.25	2.28	2.12	2.87	C XT
arbiter	1.68	1.42	1.37	1.40	1.28	1.82	XT
bw	2.03	1.80	1.80	2.87	1.62	2.33	A
clip	1.32	1.08	1.05	1.08	0.95	1.50	A
count	1.55	1.32	1.23	1.22	1.13	1.73	C
des	41.75	35.78	41.90	36.85	31.63	38.58	C
duke2	3.83	3.42	3.35	3.33	2.98	4.08	XT A
f51m	1.43	1.22	1.20	1.20	1.08	1.68	A
frg1	1.53	1.37	1.37	1.30	1.18	1.82	C
frg2	10.60	9.28	9.12	9.28	8.15	10.43	C
k2	10.17	9.17	9.25	9.10	8.10	10.35	C
pair	18.57	15.52	16.15	14.72	13.92	17.27	C
rd84	2.17	1.88	1.85	1.90	1.57	2.42	XT A
rot	8.13	6.97	6.82	6.72	6.33	7.85	C XT A
vg2	1.10	0.95	0.93	0.92	0.80	1.35	XT A
total(C)	119.07	102.26	108.28	101.55	92.32	114.74	
total(XT)	65.31	55.20	55.52	54.08	48.65	66.41	
total(A)	46.41	38.75	39.41	39.35	34.84	45.90	
total	174.27	148.89	155.52	148.74	131.13	172.56	

Table 7.3: CPU time in seconds on a Sun 3/80 used by Xmap, Amap, and XAmap for several benchmarks. Subset C is the set of benchmarks for which Chortle results are available. Subsets XT and A are subsets for which mis-pga timings are available for Xilinx cells and Actel cells.