	original files				optimized			
file	xcmap	xmap	xtmap	cpu	xcmap	xmap	xtmap	cpu
5xp1	66:5	60:9	63:5	20.7	21:3	19:3	18:3	9.0
9sym	193:23	127:37	152:23	44.2	8:3	8:3	8:3	55.4
9symml	62:5	56:7	58:5	12.2	8:3	8:3	8:3	9.1
C499	100:5	72:6	68:5	43.8	94:4	75:6	78:4	638.9
C880	156:8	102:12	140:8	98.4	170:7	118:12	170:7	160.4
alu2	196:11	129:21	149:11	121.3	90:5	97:9	80:5	70.3
alu4	329:11	251:19	267:11	235.8	260:8	216:13	224:8	313.6
apex2	2830:134	2410:271	2535:134	1313.7				
apex4	995:43	1011:46	996:43	535.2	533:5	689:9	533:5	1339.4
apex6	283:5	223:7	228:5	54.5	255:4	218:8	218:4	102.1
apex7	106:4	77:5	85:4	29.9	115:4	83:7	82:4	38.2
b9	51:3	41:4	43:3	9.9	42:3	33:3	39:3	6.8
bw	28:1	28:2	28:1	11.8	28:1	28:2	28:1	15.1
clip	263:13	207:19	228:13	75.6	59:4	51:6	47:4	44.6
count	31:5	31:5	31:5	3.6	39:3	33:4	33:3	6.6
des	1474:10	1262:13	1311:10	928.5	1320:7	1015:12	1057:7	1325.6
duke2	238:8	218:12	225:8	101.2	202:4	162:8	176:4	171.0
e64	403:16	400:16	402:16	81.1	157:3	104:6	113:3	205.0
f51m	29:4	26:5	28:4	6.4	17:3	16:3	17:3	6.3
misex1	23:3	22:3	24:3	7.5	19:2	17:3	20:2	7.0
misex2	48:3	44:3	42:3	8.1	51:3	40:4	38:3	10.1
rd73	235:18	155:34	190:18	69.9	10:2	9:2	9:2	27.8
rd84	388:42	352:113	347:42	268.1	16:3	14:3	14:3	90.8
rot	316:10	237:19	260:10	112.4	316:7	235:12	261:7	233.4
sao2	104:8	88:9	94:8	18.6	52:4	47:7	44:4	27.3
vg2	222:11	187:21	214:11	67.3	121:5	84:7	96:5	82.1
z4ml	12:3	11:3	12:3	3.9	10:2	7:2	6:2	3.7

Table 1: The columns headed with mapper names report the number of 5-input lookup tables and unit-delay estimate of delay for that mapper, either for unoptimized input (the original benchmark files) or optimized using LocalFactor and 2-column rectangle replacement. The cpu time (on a Sparcstation) is the total time for the optimization and all three mappings. Numbers are missing for optimizing apex2, because ITEM ran out of space, despite 128 Mbytes of RAM.

[5] Srinivas Devadas and Kurt Keutzer. Necessary and sufficient conditions for robust delay-fault testability of combinational logic circuits. In *Advanced Research in VLSI*, pages 221–238, MIT, 2–4 Apr 1990.

[6] Robert J. Francis, Jonathan Rose, and Zvonko Vranesic. Technology mapping of lookup table-based fpgas for performance. In *ICCAD-91*, pages 568–571, 11–14 Nov 1991.

[7] Kevin Karplus. Representing Boolean functions with If-Then-Else DAGs. Technical Report UCSC-CRL-88-28, Computer Engineering, University of California, Santa Cruz, Dec 1988.

[8] Kevin Karplus. Using if-then-else DAGs for multi-level logic minimization. In *Advanced Research in VLSI*, pages 101–118, Pasadena, CA, 20-22 March 1989.

[9] Kevin Karplus. Amap: a technology mapper for selector-based field-programmable gate arrays. In 27^{th} *Design Automation Conference*, pages 244–247, 17–21 June 1991.

[10] Kevin Karplus. Xmap: a technology mapper for table-lookup field-programmable gate arrays. In 27^{th} *Design Automation Conference*, pages 240–243, 17–21 June 1991.

[11] Kevin Karplus. Canonical if-then-else dags are path-delay-fault testable. Technical report, Computer Engineering, University of California, Santa Cruz, forthcoming 1992.

[12] Rajeev Murgai, Narendra Shenoy, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Improved logic synthesis algorithms for table look up architectures. In *ICCAD-91*, pages 564–567, 11–14 Nov 1991.

[13] Søren Søe. Variable ordering in If-Then-Else DAGs. Technical report, Computer Engineering, University of California, Santa Cruz, forthcoming 1992.

[14] Søren Søe and Kevin Karplus. Logic minimization using two-column rectangle replacement. In 27^{th} *Design Automation Conference*, pages 470–473, 17–21 June 1991.

lutheight estimate, followed by rescaling the parameters and doing five extra mappings with parameters in a small random neighborhood (see Table 1).

The generate-and-test mapper is much slower than Item's other mappers, primarily because of the cost of evaluating all the potential cells that are not used. A more directed generator that avoided generating some obviously bad cells could get similar results much more quickly.

4.2 Mapping to selector functions

Last year I presented two new algorithms [9]: Amap and XAmap, both for mapping to selector-based gate arrays, specifically, Actel's Act1 cell. The Amap algorithm tries to match the selector structure of the gate array with the if-then-else triples of the DAG. The XAmap algorithm uses Xmap to map to arbitrary three-input functions, then replaces those functions with Actel cells that implement them. These functions have not yet been reimplemented in the C++ version of Item, and so will not be reported on here.

I plan to implement generate-and-test mappers for Actel's Act1 and Act2 cells, similar to xtmap. A preliminary version was written a year ago, and looked quite promising, but has not yet been translated into C++ for inclusion in the new system.

4.3 Mapping to other cells

I plan to write generators to do generate-and-test mapping for other cells, including Quicklogic's multiple-output cell. For multiple output cells, I'll probably add three more weights for the output nodes of the cell, similar to the weights for inputs and hidden nodes.

Crosspoint's FPGA style is also an interesting one to map to, as 2-1 multiplexers are very cheap—it may turn out that just minimizing the number of edges in the if-then-else DAG and translating directly to Crosspoint's multiplexer cells will work well, even though the transistor pairs will not be used much.

5 Conclusions and Future Work

The if-then-else DAG is a powerful, flexible data structure for representing logic and multi-level logic circuits. It provides interesting new approaches for logic synthesis, including using canonical forms for highly testable circuits, local transformations for minimization, two-column rectangle replacement, and technology mapping to FGPAs.

The ITEM system is publically available for use without license fees, as long as it is not resold. It can be obtained by anonymous FTP from the site ftp.cse.ucsc.edu.

Although this paper is too short for extensive results, Table 1 gives some benchmark results for some of the techniques mentioned. For each mapper, the number of 5-input lookup tables and the unit-delay are given. Merged-CLB counts are not reported, because ITEM's merge algorithm does not take routing into account, and so the counts it gets are artificially good. The cpu time reported is for reading the BLIF input file, doing the optimization (if any), and doing all three mappings on a Sparcstation. The slowest operation is the xtmap mapper.

Item has many areas for further work, including

- better heuristics for variable ordering,
- better heuristics for finding common subexpressions, perhaps using rectangle replacement in different ways,
- technology mappers to cell libraries or cell generators,
- better delay estimators, and calibration programs for choosing estimators for different technology mappers and technologies, and
- techniques for minimizing sequential circuits.

Acknowledgements

I would like to thanks the students who have helped with ITEM over, especially Søren Søe, who implemented two-column rectangle replacement and has investigated variable ordering heuristics, and the undergraduates who did much of the early conversion work to C++: Douglas Jones, Mehrdad Parsa, J. F. Unson, John Wendt, and Thomas Wylie.

References

[1] Robert K. Brayton. Algorithms for multi-level logic synthesis and optimization. In *Design Systems for VLSI Circuits—Logic Synthesis and Silicon Compilation*, pages 197–247. Martinus Nijhoff, 1987.

[2] Michael J. Bryan, Srinivas Devadas, and Kurt Keutzer. Testability-preserving circuit transformations. In *ICCAD-90*, pages 456–459, 11–15 Nov 1990.

[3] Randal Everitt Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug 1986.

[4] Jason Cong, Andrew Kahng, Peter Trajmar, and Kuang-Chien Chen. Graph based fpga technology mapping for delay optimization. In *FPGA'92: First International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, pages 77–82, Berkeley, CA, 16–18 Feb 1992.

3.4 Two-column rectangle replacement

Although ITEM is based on a different representation of circuits and logic expressions than other minimizers, my research group has adapted techniques from other systems. For example, one of the most successful and widely-used logic minimizers is misII [1], and its most powerful technique appears to be *rectangle covering*, used for common cube extraction and kernel extraction.

We came up with a simpler, faster version of rectangle covering, which we called *two-column rectangle replacement* [14]. We use this heuristic to find common subexpressions that are hidden by the ordering of commutative expressions, similar to common-cube extraction in misII. Because two-column rectangle replacement merely reorders commutative expressions, it preserves separate support and, hence, path-delay-fault testability.

4 Technology mapping

ITEM, like most logic minimizers, has two parts: a technology-independent minimizer that tries to simplify the logic without directly considering what technology will be used to implement the circuit, and a technology mapper that tries to find a good implementation in a specific technology.

Most of the early work in ITEM was on the technology-independent minimization, using misII's mapper for mapping to cell libraries. More recent work has focussed on mappers for field-programmable gate arrays—primarily table-lookup arrays, but also multiplexer-based arrays.

Field-programmable gate arrays 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 way to configure a cell in 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. Limiting the size of the library results in inefficient use of the cells.

Item's mappers are all *faithful* mappers–ones that make no change to the underlying if-then-else DAG.

Definition 3: *A technology mapper is* faithful *if*

- *the output of each gate computes the function of some node in the if-then-else* DAG*, and*
- *the nodes corresponding to the inputs of the gate form a vertex cut set separating the output node of the gate from the primary inputs.*

4.1 Mapping to table-lookup functions

The Xmap algorithm [10] maps to the Xilinx chip and other table-lookup-based gate arrays. The mapper first maps to arbitrary f-input functions, then looks for functions that can be merged into a single Xilinx cell (the Xilinx cell allows two $(f-1)$-input functions in one cell, if the total number of inputs does not exceed f). Xilinx's 2000 Series has $f = 4$, and the 3000 Series has $f = 5$; the 4000 Series uses a compound cell that I have not yet tried mapping to.

The Xmap algorithm does an excellent job of mapping to minimize the number of lookup-tables—the only one I know to be better (mis-pga [12]) is not a faithful mapper, but does significant higher-level optimization.

For delay minimization, the main contenders have been Chortle-d [6] and Dagmap [4]. The core of Dagmap has been implemented in Item as the *lutheight* delay estimator, and the mappings it produces can be examined with the Xcmap mapper. This algorithm usually produces slightly smaller delays than chortle-d, and much smaller ones than xmap or mis-pga. Although the xcmap (Dagmap) algorithm is optimal for trees, on one highly reconvergent example xmap produces a circuit with fewer levels of lookup tables.

I have recently been experimenting with a new mapper (xtmap) based on a generate-and-test paradigm. This algorithm generates many possible single-cell implementations for a node, then chooses one based on some simple heuristics. After choosing a cell, the algorithm is applied recursively to the inputs for the cell.

Cells are evaluated based on the set of inputs and the set of if-then-else nodes "hidden" inside the cell. The value of a cell is computed as the sum of several weights: Each input that already has a cell mapping contributes weight a, inputs that aren't already mapped contribute weight $b + c/\text{fanout(node)}$ each. Similarly, hidden nodes that have already been mapped contribute d each, while unmapped ones contribute $e + f/\text{fanout(node)}$ each. Finally, $g \max\text{lutheight}(input)$ is added.

The heuristics can be tuned by either random search in the parameter space (slow, but effective), or by learning them from a known good mapping. The benchmark results for xtmap in this paper are obtained by learning the parameters from an xmap mapping, then increasing the penalty for delay until the unit-delay estimate for the output of xtmap matches the

any particular cell library (or that are parameterized and easily tuned for different technologies), and that roughly approximate the cost or speed obtained by a technology mapper. Technology-independent *delay* estimates are hard to come up with, and so most early research concentrated on *size* minimization, leaving the delay minimization to the technology mapper.

The usual way to estimate the area for a network of gates is to estimate the cost for each gate and sum the estimates. The most popular gate area estimators are the number of literals in sum-of-products form and the number of literals in the factored form [1, page 235]. The literal count corresponds closely to the number of transistor pairs needed to implement the function as a static cMOS gate, and is an excellent area estimator if the mapper does not change the decomposition of the circuit. The estimate is not as good when the mapper splits or merges gates, but the predictions can be improved by subtracting the number of gates from the sum of literals [8].

The standard measures described above are useful when a network has been decomposed into gates, but are not directly applicable to a network described as an if-then-else DAG with multiple roots. I have experimented with several new estimators, and found the following to be particularly useful for area estimation:

edges, the number of non-constant pointers in the DAG, is good for mapping to FPGAs (both table-lookup and multiplexer gates).

count, a recursively defined function that attempts to match the values of the estimator (literals(factored form)+ outputs− gates), is good for the predicting the behavior of the misII technology mapper.

Estimating delays seems to be harder than area estimation–the best predictors I have found are the following:

height, the longest path from a root to a leaf, is an adequate, but not good predictor of the delay in misII's mapper. Better delay estimates are needed before higher-level optimizations for delay are feasible.

lutheight, the minimum height needed for an implementation as k-input lookup tables, using Cong's algorithm [4], is an excellent estimator of the delay predicted by a unit-delay model, but placement and routing affect the delay enormously, and we have not yet done experiments to see how well the real delay after place-and-route is predicted.

3.2 Preserving testability

Any transformation of an if-then-else DAG that preserves the separate-support, distinct-cases, and no-constant-if conditions, will preserve path-delay testability, although not necessarily the test set [11]. Two of Item's transformations preserve the conditions: conversion to canonical form [7] and two-column rectangle replacement [14]. Furthermore, LocalFactor (see Section 3.3) can be constrained to preserve them. The DeIf transformation, which replaces the if-then-else gates with primitive gates, cannot preserve the separate-support condition.

The faithful mappers (see Section 4) do not preserve the separate support condition, but any path-delay-fault test set for the if-then-else DAG will also be a test set for the mapped circuit.

The most effective testability-preserving transformation available in ITEM appears to be conversion to canonical form with a different variable orderings [13], optionally followed by two-column rectangle replacement [14] (see also Section 3.4).

3.3 LocalFactor

My initial work in logic minimization used local transformations applied to all portions of the DAG, generally in a depth-first traversal from the outputs. I came up with two sets of transformations: Printform and LocalFactor.

The Printform transformations [8] preserve separate support, and so preserve path-delay-fault testability, but are not good at minimizing circuit area. The reason they are not good circuit minimizers is that they were originally designed to minimize the size of printed Boolean expressions, not multi-level circuits.

The LocalFactor transformations started as a rather *ad hoc* collection of transformations that did an adequate job of minimizing circuit area [8]. Recently, I reorganized the transformations to include the Printform transformations, conversion to canonical form, and several other sets of transformations.

The LocalFactor routine applied to a node in an if-then-else DAG first applies itself recursively to the children of the node, then applies various transformations to the triple of transformed children. Whenever a transformation is applied, it establishes the logical equivalence of two different representations of a function. These equivalent representations are linked together in an *equivalence ring*. After applying all relevant transformations, the representations in the equivalence ring are evaluated using some cost function, and the cheapest one returned.

By turning on and off different transformation sets, I found that the most valuable transformations are the conversions to canonical form.

A weaker restriction, that the variables of the **if**-part be disjoint from those of the **then**- and **else**-parts, is not enough to make the if-then-else DAG canonical, but is all that is needed for path-delay-fault testability (see Section 2.2). This weaker restriction is referred to as the *separate-support condition*.

2. *Systematic-negation condition:* A choice must be made between the equivalent DAGs (**if** a **then** b **else** c) and (**if** a' **then** c **else** b) and between (**if** a **then** b **else** c) and (**if** a **then** b' **else** $c')'$. Item requires that **if**- and **then**-parts of a node be pointers labeled **plus**, with negation allowed only for the **else**-part or the pointer to a root of the DAG. This corresponds to Bryant's choice of variables as node labels (never negations of variables).
3. *Distinct-cases condition:* The **then**- and **else**-parts of a node must be distinct Boolean functions.
4. *No-constant-if condition:* Triples whose **if**-part points to TRUE are prohibited, and should be replaced by the **then**-part.
5. *No-two-constant condition:* Triples in which both the **then**- and **else**-parts point to TRUE (with either plus or minus labels) are prohibited. The triple should be replaced by an appropriately labeled pointer to the **if**-part or to TRUE.
6. *No-common-cut condition:* In the triple (**if** a **then** b **else** c), b and c must not share both **then**- and **else**-parts. If $b = (\textbf{if}\ b_a\ \textbf{then}\ b_b\ \textbf{else}\ c_c)$ and $c = (\textbf{if}\ c_a\ \textbf{then}\ b_b\ \textbf{else}\ c_c)$, then the correct representation is (**if** (**if** a **then** b_a **else** c_a) **then** b_b **else** c_c). If $b = (\textbf{if}\ b_a\ \textbf{then}\ b_b\ \textbf{else}\ b_c)$ and $c = (\textbf{if}\ c_a\ \textbf{then}\ b_c\ \textbf{else}\ b_b)$, then use (**if** (**if** a **then** b_a **else** c_a') **then** b_b **else** b_c).
7. *No-collapsed-cut condition:* In the triple (**if** a **then** b **else** c), b must not contain c as a **then**- or **else**-part. If $b = (\textbf{if}\ b_1\ \textbf{then}\ b_b\ \textbf{else}\ c)$ or $b = (\textbf{if}\ b_2\ \textbf{then}\ c\ \textbf{else}\ b_c)$, then the DAG should be changed to (**if** (**if** a **then** b_1 **else** FALSE) **then** b_b **else** c) or (**if** (**if** a **then** b_2 **else** TRUE) **then** c **else** b_c). If c is a constant (TRUE or FALSE), then this restriction amounts to choosing left-associativity for commutative AND or OR operations. The symmetric test for $c = (\textbf{if}\ c_1\ \textbf{then}\ c_b\ \textbf{else}\ b)$ or $c = (\textbf{if}\ c_2\ \textbf{then}\ b\ \textbf{else}\ c_c)$ is also needed.

2.2 Canonical forms are highly testable

There has recently been much attention on the synthesis of robustly path-fault-delay testable circuits [5, 2]. Most of this work has focussed on transformations applied to two-level circuits and to multi-level circuits derived by algebraic factoring.

Synthesis for testability consists of two parts: generating a testable circuit from the initial logic equations or circuit, and applying testability-preserving transformations to improve the circuit. The canonical form presented in Section 2.1 is an excellent starting point for synthesis for testability. The canonical form can be shown to be robustly path-delay-fault testable.

Devadas and Keutzer proved that complete robust path-delay-fault tests are also complete tests for single and multiple stuck-at faults [5, Theorem 5.6]. Although their proof was given only for circuits composed of primitive gates, the same proof works for more complex gates, as long as you restrict the locations that faults can be inserted to the inputs and outputs of the complex gates (faults internal to the gates are not necessarily found). Because Item's canonical forms are robustly path-delay-fault testable, testability for single and multiple stuck-at faults is easy.[1]

Not all the properties of the canonical form are needed for the proof of testability. In particular, the variable ordering condition can be relaxed to the separate-support condition, and some of the other conditions can be discarded entirely without losing testability. Transformations that preserve the critical properties can be applied to reduce the area or delay of the circuit further.

The test set for path-delay-fault testing can be quite large, even when the if-then-else DAG is small. Tree circuits have few paths, but even a few levels of reconvergent fanout can cause exponential increases in the number of paths. Some functions (such as multipliers) cannot be efficiently implemented without reconvergent fanout.

3 Logic minimization

3.1 What is minimized?

When doing logic minimization, the first question is "what exactly is being minimized?" The goal of minimization is to reduce the area, power, delay, or testing difficulty of the final circuit after technology mapping. For technology-independent minimization to work, we need measures that are not dependent on

[1] In most implementations of multiplexers, there are two paths from the control input to the output: one inverting, and one non-inverting. Because this splitting of the path occurs inside the complex gate, a fault on one internal path may not be detectable, and so some reviewers have objected to my use of "robustly path-delay-fault testable". For many technologies (such as table-lookup FPGAs), the primitive gate model does not do any better job of testing internal faults, and so it is equally reasonable to model the complex gates as atomic.

ITEM: an If-Then-Else Minimizer for Logic Synthesis

Kevin Karplus*

Board of Studies in Computer Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064

Abstract

This paper introduces ITEM, the combinational logic minimization program developed at the University of California, Santa Cruz. ITEM is useful for generating highly testable circuits, because canonical if-then-else DAGs are robustly path-delay-fault testable, and often produce samll, fast circuits. Several of our transformations preserve testability, including the Xmap and Amap technology mappers for field-programmable gate arrays.

The paper includes the definitions of if-then-else DAGs and canonical forms, introduces a new technology mapper for table-lookup FPGAs, briefly mentions the main transformations available in ITEM, and provides a table of benchmark results for mapping to 5-input tables using various transformations.

This paper also introduces a new technology mapper based on a generate-and-test paradigm.

1 What is ITEM?

This paper introduces ITEM, the combinational logic minimization (and technology mapping) program developed at the University of California, Santa Cruz over the last six years. The algorithms and data structures were originally developed in C, but in the past two years ITEM has been completely redesigned and rewritten in C++ for easier modification and extension.

ITEM uses directed, acyclic graphs to represent logic functions. The only operators used are the if-then-else operator (2-to-1 multiplexer) and negation. These if-then-else DAGs are a simple generalization of Bryant's Binary Decision Diagrams [3].

Definition 1: *The* if-then-else *operator is a ternary Boolean function, with* (**if** a **then** b **else** c) *defined as* $ab + a'c$ *or, equivalently,* $(a + c)(a' + b)$.

Definition 2: *An* if-then-else DAG *is a ternary directed acyclic graph in which*

- *each leaf is labeled with the constant* TRUE *or a variable,*
- *each edge is labeled with a polarity, either* **plus** *or* **minus**, *and*
- *each internal node has three out-edges pointing to* **if**-, **then**-, *and* **else**-*parts.*

There may be multiple root nodes for an if-then-else DAG, corresponding to different principal outputs of a circuit. I sometimes refer to a node as a DAG, meaning the entire subDAG rooted at that node.

I have found the if-then-else DAG to be a marvelously flexible representation scheme. It can be used to represent circuits directly (a circuit built from two-input gates can be represented with one if-then-else triple per gate), and it can be used for general manipulation of logic functions. Canonical forms have turned out to be useful not only for verification, but also as small circuits for implementation.

2 Canonical forms for if-then-else DAGs

2.1 Definition of canonical form

A representation is *canonical* if any two expressions that are logically equivalent are identical. For example, if $ab + ab'$ is represented differently from a, then the representation is non-canonical.

To make if-then-else DAGs canonical, we must place some restrictions on the subDAGs allowed in the **if**-, **then**-, and **else**-parts of every internal node of the DAG. We can show that imposing the following seven restrictions defines a canonical form [7, 8].

Of the seven restrictions, the first five are modified versions of the corresponding restrictions in Bryant's canonical form for binary decision diagrams [3]. The last two restrictions differentiate between the two canonical forms. Essentially the same algorithm can be used for converting to either Bryant's canonical form or the new form [7].

1. *Variable ordering condition:* A total ordering is imposed on the variables, and all the variables in the **if**-part must be earlier in the order than all variables in the **then**- and **else**-parts.

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