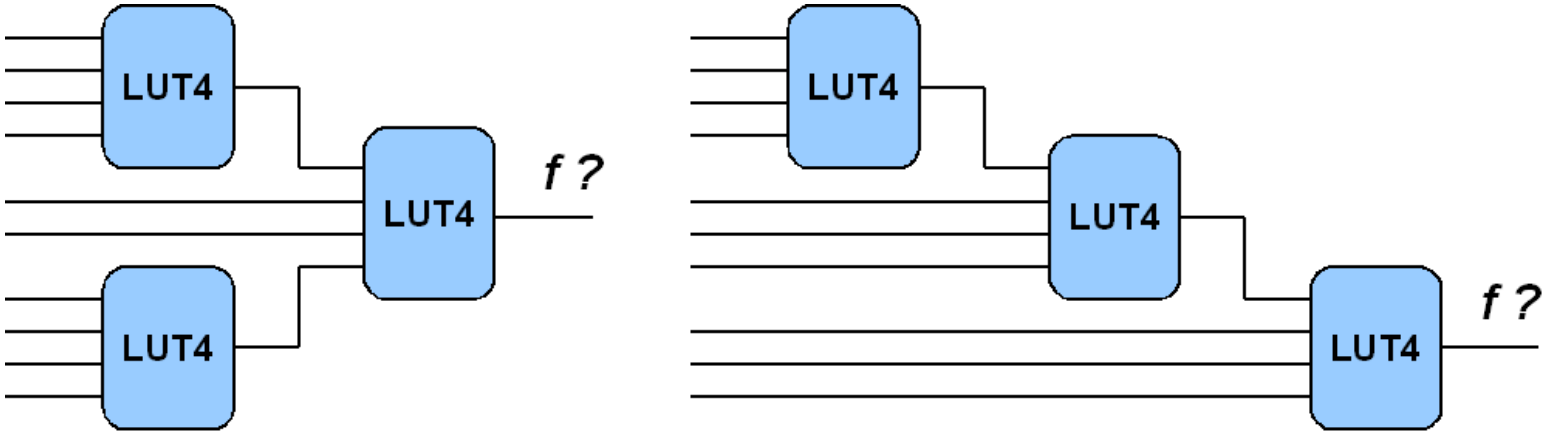
Generating efficient libraries for use in FPGA re-synthesis algorithms

Andrew Kennings, University of Waterloo
Alan Mishchenko, UC Berkeley
Kristofer Vorwerk, Val Pevzner, Arun Kundu, Actel Corporation

The problem (1)

- Can a given cone of logic (implementing a logic function ***f***) be implemented by some topology of Look-Up Tables (LUTs)?



The problem (2)

- The problem of fitting a logic function into a topology of LUTs is one particular version of ***Boolean Matching***.
  - *The solution to the problem must not only return a yes/no result, but also the necessary input connections and LUT configuration bits.*

- Important problem to address - Boolean Matching can be used extensively within FPGA re-synthesis algorithms; e.g.,
  - *Post-technology mapping for area-oriented re-synthesis (reduce LUT count w/o harming timing);*
  - *Post-placement for performing timing-oriented re-synthesis (improve timing by re-implementing cones of logic).*
  - *…*

Possible matching strategies (1)

- Structural-based techniques:
  - *Converts the cone of logic back into a 2-bounded network (e.g., an AIG) and applies remapping;*
  - *Efficient and fast for small cones of logic, but structurally biased.*

- SAT-based techniques:
  - *Formulates the problem as a SAT instance; Variables represent the configuration bits and the input permutations;*
  - *Guaranteed to find a match (if possible), but slow and limited in problem size.*

Possible matching strategies (2)

- Decomposition-based techniques:
  - *Apply known algebraic or Boolean decomposition techniques such as Roth-Karp or DSD to "break" or "decompose" the logic function into smaller sub-functions implementable by individual LUTs;*
  - *Speed depends on the logic function being decomposed; can be fast or slow. Effectiveness depends on the decomposition technique used and heuristics implemented to speed up decomposition.*

- Class-based techniques:
  - *Determine a canonical representation for the logic function and look-up the canonical form in a pre-computed library;*
  - *Very fast (efficient canonicalization routines), but requires a library against which matching can be performed. Results are only as good as the library. Can be memory intensive to store the library and can be difficult to create a library.*

Our proposal (1)

- No single matching strategy appears to satisfy both the goals of being efficient and effective.

- We propose to use a combination of class- and decomposition-based techniques which is hopefully more efficient and effective vs. one single method.
  - *Use decomposition off-line to create a library of decompositions for commonly seem logic functions;*
  - *During matching, use a class-based technique against the generated library (decomposition then becomes a "fall-back" strategy).*

- The library (plus class-based matching) effectively becomes a useful filter which uses fast look-ups to quickly decompose very common logic functions.

Our proposal (2)

- In our proposed method, several questions must be answered;
  - *What are commonly occurring logic functions (i.e., what logic functions should be represented, decomposed off-line and stored in the library)?*
  - *How much memory is required to store the library (i.e., how large does the resulting library need to be in order to store all necessary functions and decompositions – can a useful library even be computed for LUT structures)?*
  - *Etc…*

- This talk will describe some analyses and techniques that will show that it is, indeed, feasible to generate and use libraries and class-based matching within an FPGA re-synthesis environment.

Common logic functions (1)

- ***Observation #1***: *Only a small set of n-input logic functions (NPN equivalence classes) exist in real designs.*
  - *It is only necessary to store (in the library) those logic functions (and their decompositions)* ***which occur in real circuits****.*

- To confirm, all *n*-input logic functions were computed over a large set of FPGA designs with the following results (***5 <= n <= 9***):

Number of Inputs (n)	# Logic Functions	# NPN Classes
5	7768377	3269
6	18814641	34225
7	50239975	271646
8	146678254	2317679
9	165876500	7145748

the number of NPN classes is actually small (for a particular value of n) vs. what it might be!

Common logic functions (2)

- ***Observation #2**: Some functions (equivalence classes) occur more frequently than others.*
  - *It is only necessary to store (in the library) those logic functions (and their decompositions) **which occur frequently**.*

Number of Inputs (n)	#Equivalence classes to cover % of all logic functions		
	95%	90%	80%
5	87(2.7%)	55(1.7%)	42(1.3%)
6	638(1.9%)	311(0.9%)	207(0.6%)
7	7322(2.7%)	2437(0.9%)	1313(0.5%)
8	101368(4.4%)	26956(1.2%)	11399(0.5%)
9	991073(13.9%)	215899(3.0%)	70242(1.0%)

- So, for example, while 3269 NPN classes are required to cover 100% of all 5-input logic functions, only 42 NPN classes are required to cover 80% of all 5-input logic functions.

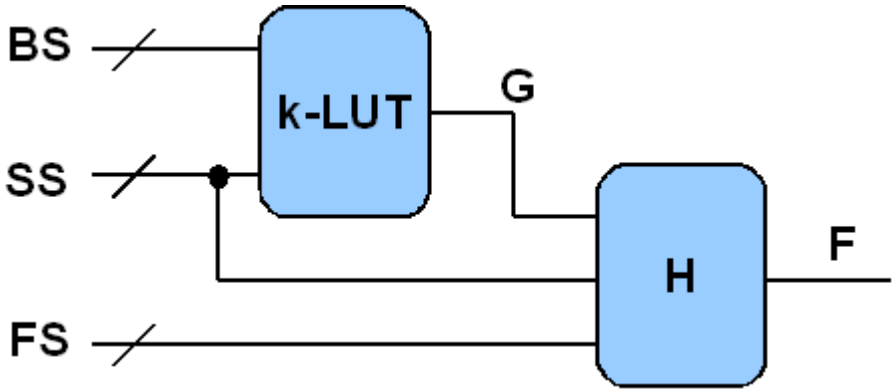
Common logic functions (3)

- From the analysis of logic functions which occur in real designs, it becomes apparent that not that many logic functions need to be represented (present) in the library;
  - *This means that the library might not consume too much memory, but will still provide an effective way to search for decompositions for common logic functions.*

- We note that analysis over a different benchmark set might yield different results, but we believe that the more common logic functions we have found will always tend to be common across different benchmark suites!

Library generation (1)

- Although the number of logic functions is reasonable, it is still necessary to generate and store the decompositions for each logic function.
  - *Structures of LUTs for the required logic functions are found, for example, using Roth-Karp decomposition.*



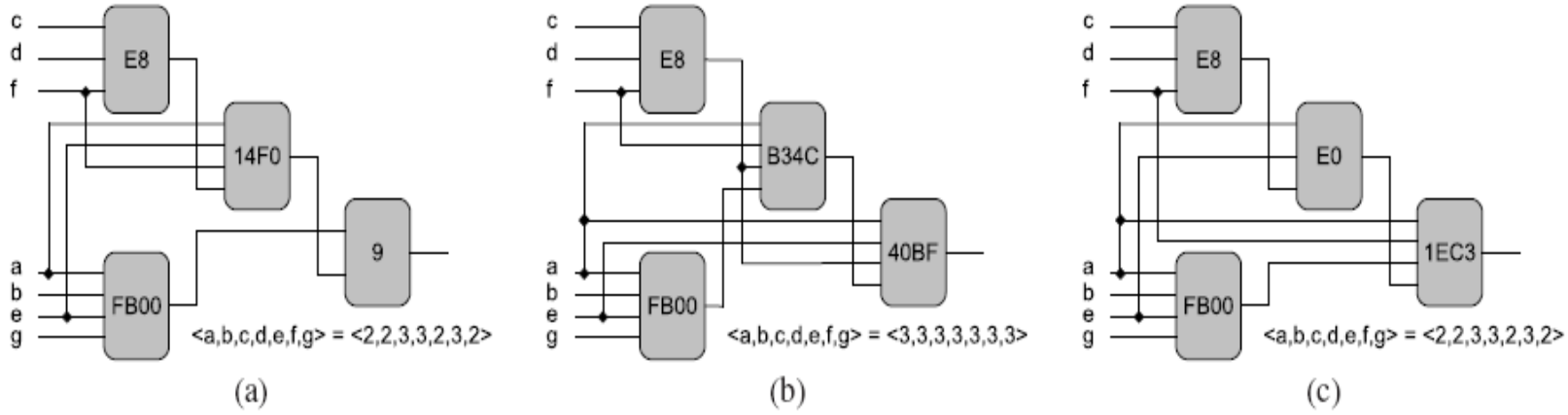
  - *Decomposition is enumerative when generating structures for the library to find all input combinations (important for timing).*

Library generation (2)

- For a given logic function, it is possible that there are ***many*** possible decompositions;
  - *Some decompositions are repetitive and only waste memory if stored;*
  - *Some decompositions dominate others if timing is approximately taken into account.*
- Define the ***timing profile*** of any decomposition ***A*** are the ordered vector of depths from each input pin of ***A*** to the output of ***A***;
  - *i.e.,* ***timing_profile(A) = <depth(input(0)), depth(input(1)), … , depth(input(n-1))>.***
- Given two decompositions ***A*** and ***B*** for a function ***f***, we say that ***A dominates B*** if:
  - *If* ***area(A) < area(B)****, or*
  - *If* ***area(A) = area(B)*** *and* ***timing_profile(A) <= timing_profile(B).***
- Dominated decompositions can be discarded from the library.

Library generation (3)

- The use of timing profiles is illustrated as follows for a *7*-input logic function *f*;
  - *If we assume that decomposition (a) is computed first, then decompositions (b) and (c) can be discarded since they offer no benefits in terms of area or delay.*



  - *The main point is that we can toss decompositions that do not look promising.*

- Using Roth-Karp along with the idea of dominated structures leads to an average of ***~2.7 to ~5.8*** structures stored per logic function for ***5 <= n <= 9*** input logic functions.

Memory requirements (1)

- At this point, we know that:
  - *we only need to store a small set of logic functions, and*
  - *The number of decompositions for each logic function is, on average, small.*

- The amount of memory required to implement a library can now be computed;
  - *Need to store decompositions which, in turn, are composed of LUTs;*
  - *For a LUT it is necessary to store (1) its configuration bits and (2) identifiers for its inputs.*

- Analysis shows that for a single k-LUT (3 <= k <= 6), the following number of bits (# unsigned words) are required:

Size of LUT	3	4	5	6
#Bits	23	32 / 36	57	94
#Unsigned words (32 bits)	1	1 / 2	2	3

Memory requirements (2)

- Estimate of the total library size for different function coverage (based on #functions, #decompositions and #words to store a LUT):

- E.g., for a LUT4 architecture:

Number of inputs (n)	Percentage coverage of logic functions			
	100% (#bytes)	95% (#bytes)	90% (#bytes)	80% (#bytes)
5	290K	7.7K	4.9K	3.0K
6	3.9M	72.7K	35.4K	16.9K
7	51.0M	1.4M	457.8K	162.5K
8	409M	17.9M	4.8M	1.1M
9	1123M	155.7M	33.9M	4.6M

- Quite reasonable memory requirements to cover a large number of logic functions.

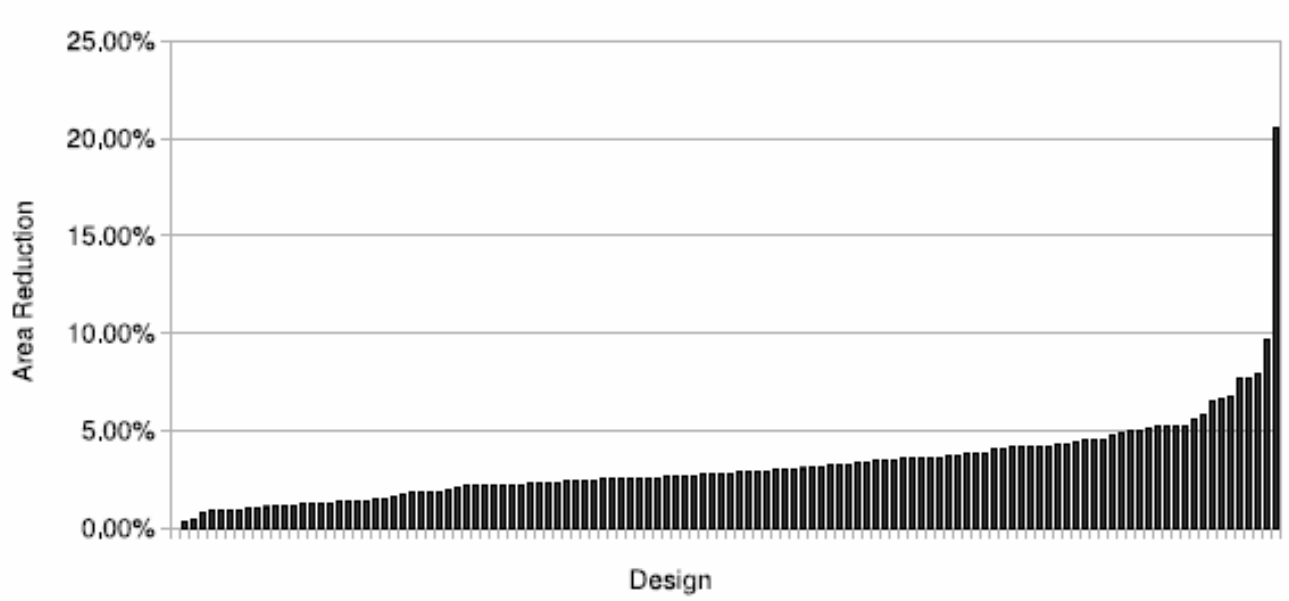
Numerical results (1)

- Performed some numerical tests to determine the effectiveness of our proposed matching technique.
  - *Implemented an area-oriented re-synthesis algorithm used after technology mapping along with a set of ~100 FPGA designs all subjected to technology independent logic optimizations, converted to subject graphs and mapped to LUT4 architecture.*
  - *When a library is provided, we first attempt to match to decompositions in the library. If no match is found, then we resort to decomposition.*

- Wanted to determine:
  - *How effective was the area-oriented re-synthesis (**baseline for other tests**)?*
  - *How useful is the library at reducing run-time (vs. decomposition) and what is the penalty paid for less comprehensive libraries?*

Numerical results (2)

- Area savings from the area-oriented re-synthesis (3.2% savings on average with a maximum of 20%).



- Not the important result; this is just for comparison purposes.

Numerical results (3)

- Performed area-oriented re-synthesis with libraries offering less and less coverage.
  - *Note that the quality of result was not impacted by this experiment; we simply changed how we performed matching (match to library vs. on-the-fly decomposition done the same way as during library generation).*

Library Coverage	CPU	Library Hit Ratio
99%	1.71	0.96
98%	3.44	0.94
97%	3.94	0.92
96%	4.66	0.90
95%	5.33	0.89
90%	11.50	0.82

- Incomplete libraries are effective, but there is a run-time penalty for less comprehensive coverage.

Numerical results (4)

- Changed decomposition algorithm to make it more restrictive (and therefore faster – specifically allowed only 1 shared set variable during decomposition).

Library Coverage	CPU	Library Hit Ratio	QOR
99%	1.47	0.96	3.17%/20.5%
98%	1.76	0.94	3.17%/20.5%
97%	1.89	0.92	3.16%/20.5%
96%	1.99	0.90	3.16%/20.5%
95%	2.02	0.89	3.16%/20.5%
90%	2.87	0.82	3.16%/20.5%

- Faster decomposition vastly improves run-time w/o too much loss in quality.
- Libraries still beneficial in terms of run-time and analysis showed that most of the matching was still done against the library.

Summary

- Proposed a technique for performing Boolean matching to LUT structures which uses a combination of decomposition-based and class-based methods.

- Showed libraries (and class-based matching) are feasible for FPGAs by doing the following:
  - *Considering only those logic functions that occur **frequently** in **real circuits**, and*
  - *Pruning LUT structures which are **dominated** (via area or timing) by other LUT structures.*

(Extra) How decompositions are stored and parsed prior to use

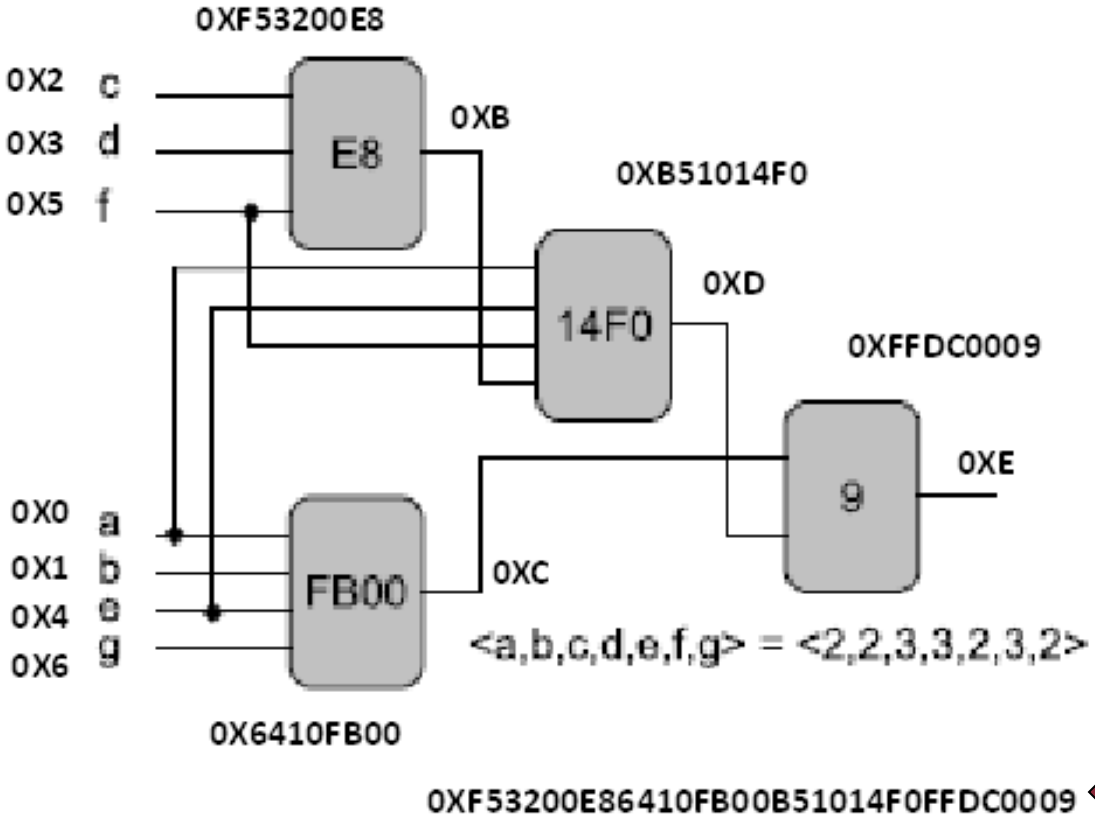
- Decompositions are recorded into a text-based library file:

```
0000000700077B77 0107(e,f,8488(a,b,c,d),E(c,d)) 0B07(a,b,FEE0(c,d,e,f),4(c,d))
500F00000000722F 1909(e,f,BC(a,c,d),2(a,b)) 002F(a,b,BC(a,c,d),E6(e,f,BC(a,c,d))) 1C03(b,f,BC(a,c,d),2C(a,e,BC(a,c,d)))
00000F0F65FB65FB 101F(c,e,f,9A04(a,b,c,d))
1818181818F81FFF 18FF(a,b,c,FFD8(c,d,e,f))
000000000A5027FF 1(f,EC70(a,d,e,F0D8(a,b,c,e))) 0853(d,e,F0D8(a,b,c,e),3FA0(a,d,e,f)) 1(f,DEB0(a,c,e,F0D0(a,b,d,e)))
444422226FF6F66F 462F(a,b,f,F096(c,d,e,f))
0F0F1510F0F0EAEF 5A69(c,e,f,E5E0(a,b,c,d)) 3C4B(b,c,f,FF98(a,c,d,e)) 1EC3(a,e,f,CC74(b,c,d,e))
00000088000F77FF 1107(d,e,f,F088(a,b,c,e))
...
```

- Easy to modify/augment with additional structures later on (given more designs)…

(Extra) Sample of memory efficient storage of a decomposition

- Storing a decomposition means storing a vector of information about the LUTs in the decomposition:



Final encoding of structure (4 unsigned words)