Efficient Orthonormality Testing for Synthesis with Pass–Transistor Selectors

Michel Berkelaar, *michel@es.ele.tue.nl*
Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, the Netherlands
Lukas P.P.P. van Ginneken, *lukas@synopsys.com*
Synopsys Inc., 700 East Middlefield Road, Mountain View, CA 94043

Abstract

This paper presents the mapping problem for pass transistor selector mapping, which has not been addressed before. Pass transistor synthesis is potentially important for semi– or full–custom design techniques, which are increasingly attracting attention. Pass transistors have the advantage that fewer transistors are needed, and that circuits with high fanin and small delay can be constructed. Technology mapping approaches in the existing literature cannot handle these selectors, due to the restriction of 1–hot encoding of the control signals. We present a new algorithm to address this problem, which is based on the novel idea of a general Boolean Oracle. Our oracle is based on ATPG techniques, and compared to BDDs, the oracle has the advantage that failure to complete only affects optimization locally, and does not hinder optimization elsewhere in the logic. A limitation of BDDs is that it is difficult to complete the algorithm if a BDD grows too large. The experimental results show up to 82% improvement in transistor count for the MCNC combinatorial multi–level examples.

1 Introduction

In this paper a *selector gate* is a gate with n pairs of inputs. Each input pair has a *data* and a *control* input. The output of the selector gate is defined only for the case that exactly one of the n control inputs is 1. In this paper we call a set of signals in which exactly one is 1 for any input vector *orthonormal*. A formal definition can be found in section 2. In case of orthonormal control inputs, the (single) output of the selector gate will reflect the value of the data input associated with the control input that is 1. A related type of gate is the *multiplexer gate*. Multiplexers have data and control inputs too, but they do not have a one–to–one correspondence of data and control inputs. In most cases multiplexers have fewer control inputs than data inputs. Some kind of decoding of the values on the control inputs determines which data input controls the multiplexer output. See Figure 1 for an example of a 4 input selector and a 4 input multiplexer with 2 control inputs. In this paper we will only discuss the problem of using selectors to implement logic.

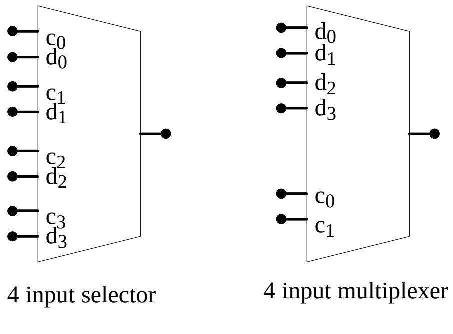


Figure 1: difference between selector and multiplexer

The problem of using the selectors which might be available in a standard cell library during technology mapping is a fundamental one. All technology mappers that the authors are aware of require that all standard cell library elements without memory can be described by a single (possibly multi–output) Boolean function. This Boolean function is internally represented by, for example, a network of 2–input *nands* and inverters (e.g. the MIS mapper [DET87]), or a BDD (e.g. CERES [MAI90]), or even a truth table (e.g. the BooleDozer mapper). This function is supposed to hold for all possible input values. Selectors also perform a Boolean function from inputs to output (a simple *and–or* function), but this function is only defined for those cases where the control inputs are orthonormal. The actual behavior for other control input values of a selector gate is implementation dependent. The pass transistor implementation of Figure 2 will have a high output impedance when all control inputs are 0, and produce some intermediate value with a high short circuit current flowing when 2 control signals are 1 with conflicting corresponding data input signals. In practical implementations it might be necessary to add an inverter or buffer to the output of a pass transistor circuit to restore the logic values, but it still remains a very efficient implementation. An introduction to pass–transistor logic can be found in [WHI81].

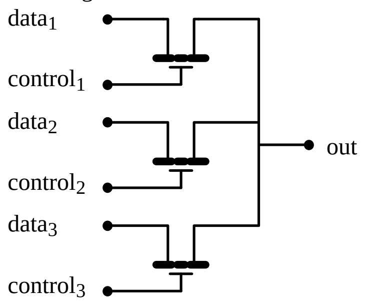


Figure 2: 3 input pass transistor selector

All of this implies that we can only map parts of a circuit to a selector after we have made sure that the control signals will be orthonormal. This function cannot be performed by a 'normal' technology mapper. Yet, selectors can be very efficient implementations for some pieces of logic, and are, therefore, present in some standard cell libraries.

This paper will describe an approach to identify those parts of circuits that can be mapped to selectors.

This paper is the first one on the topic of mapping to selectors that the authors are aware of. A lot of attention has been given to mapping circuits to multiplexer based FPGAs (e.g. [ZHU93] [LAN93]), but the problem of the requirement of orthogonal control signals does not occur here. Furthermore, we are not trying to implement a circuit by means of selectors only. We only want to replace the parts where this is "natural" and does not require the introduction of extra logic nor substantial restructuring of the logic.

This paper is organized as follows. Section 2 introduces the concept of a Boolean Oracle. We use this oracle to answer questions about the orthonormality and orthogonality of signals. In section

3 of this paper we describe how we identify parts of circuits that can be replaced by selectors. In section 4 we show how some of the selectors we have identified in section 3 can be merged into fewer but larger selectors. Section 5 presents results on selectors found in the set of MCNC combinatorial multi–level examples. Section 6 contains the conclusions.

2 The Boolean Oracle

To be able to use selectors to implement parts of logical circuits, we need to be able to answer questions about the orthogonality and orthonormality of sets of signals. In the paper we assume that we have a *Boolean Oracle* to answer our questions. To define what kind of questions we expect the Boolean Oracle to answer, we need some definitions. The word *circuit* will refer to a combinatorial Boolean circuit consisting of *gates* connected by *signals*.

Definition 1: The set $\mathcal{S}$ is the set of all signals in a circuit.

Definition 2: The set $\mathcal{N}$ is the set of all *nands* in a circuit.

In the following definitions we use the notation $\mathcal{S}^*$ to indicate the power set of $\mathcal{S}$.

Definition 3: The function $Fanin : \mathcal{N} \rightarrow \mathcal{S}^*$ gives the set of input signals of a *nand*.

Definition 4: $\mathcal{V}$ is the set of all possible primary input vectors for a circuit. For a circuit with n primary inputs without input don't cares, $|\mathcal{V}| = 2^n$. If there are don't care conditions on the inputs, $|\mathcal{V}| < 2^n$.

We will use statements like $\exists_{v \in \mathcal{V}}$ *formula* or $\forall_{v \in \mathcal{V}}$ *formula*, even though v might not appear in *formula*. But, as we limit ourselves to combinatorial logic circuits, other signals values s in the circuit, which do appear in *formula*, depend directly on v, formally requiring a notation like $s(v)$. Where this does not lead to ambiguity, we will simply write s.

Definition 5: A set of signals S in a Boolean network is called *orthogonal* when

$$\forall_{v \in \mathcal{V}}\left(\forall_{s \in S}\ \left(s = 1 \Rightarrow \forall_{s' \in S \setminus \{s\}}\ (s' = 0)\right)\right)$$

or: at most one of the signals is 1 for any input vector applied to the circuit.

Definition 6: A set of signals S in a Boolean network is called *orthonormal* when it is orthogonal and

$$\forall_{v \in \mathcal{V}}\left(\exists_{s \in S}\ \ s = 1\right)$$

or: exactly one of the signals is 1 for any input vector applied to the circuit.

In the rest of the paper, $\wedge$ will indicate the Boolean *and* operation.

Definition 7: The function $All\_1 : \mathcal{S}^* \rightarrow \{0,\ 1\}$ can be defined for $S \subseteq \mathcal{S}$ by:

$$All\_1(S) = \begin{cases} 0 & \forall_{v \in \mathcal{V}}\left(\bigwedge_{s \in S} s = 0\right) \\ 1 & \exists_{v \in \mathcal{V}}\left(\bigwedge_{s \in S} s = 1\right) \end{cases}$$

$All\_1(S)$ indicates whether or not the signals $s \in S$ can all be 1 for some input vector.

Definition 8: The function $All\_0 : \mathcal{S}^* \rightarrow \{0,\ 1\}$ can be defined for $S \subseteq \mathcal{S}$ by:

$$All\_0(S) = \begin{cases} 0 & \forall_{v \in \mathcal{V}}\left(\bigwedge_{s \in S} \bar{s} = 0\right) \\ 1 & \exists_{v \in \mathcal{V}}\left(\bigwedge_{s \in S} \bar{s} = 1\right) \end{cases}$$

$All\_0(S)$ indicates whether or not the signals $s \in S$ can all be 0 for some input vector.

A Boolean Oracle can calculate the value of All_1 or All_0. It could be implemented in several ways:

1. By exhaustive simulation of the circuit. This might be very time consuming, and is not a realistic option for circuits with many primary inputs.
2. By using BDDs. BDDs are known to grow exponentially for certain types of circuits, but behave reasonably for many practical circuits.
3. By using a test pattern generator.

We have opted for the last implementation. We use the package basically by setting the signals in a set $S \subseteq \mathcal{S}$ to fixed 0 or 1 values, and then ask the test pattern package to come up with an input pattern that will generate these internal signal values. This uses only the justification capability of the test pattern package. There are 3 possible answers:

1. If an input pattern can be found, we have set the signals to valid values.
2. If no input pattern can be found because of inconsistencies, we have set the signals to values which can never occur.
3. The test pattern package cannot answer the question conclusively because it would take too much time.

The test pattern package we use is TGFS [KUN90] [KUN92], which is part of the IBM BooleDozer[1] logic synthesis system. The speed of this test pattern package allows us to consult the Boolean Oracle many times.

A very important reason for choosing to implement the Boolean Oracle by using TGFS is the graceful way in which it fails, as compared to BDDs. If we had decided to use BDDs, we would not have been able to reason about circuits for which the BDD representation blows up. The process will run out of memory, or hit a memory limit specified to the BDD package, and we will fail to produce results for that circuit. A test generator will not run out of memory, but it might use a lot of time. However, TGFS allows one to specify the amount of effort that should be spent on a specific question. This means that the oracle may sometimes respond to a question with the answer “don't know”. In the context of this paper, this means that sometimes we cannot be sure whether a selector can be used, and have to decide not to do it. But in other places in the same circuit we may still be able to decide. For the MCNC benchmarks used in the results section of this paper, the oracle was always able to give a decisive answer. Any BDD packages would have had a hard time with the multiplier of circuit C6288, and BDD packages without dynamic variable ordering would have found other circuits difficult too [RUD93].

3 Identifying selectors

We will look for selectors in the logic in three basic steps:

1. Find parts of the circuits where the basic *and–or* function of the selector is performed. These parts are called *candidates*.
2. In each of the candidates, try to identify sets of signals which are orthogonal, and, therefore, might serve as control inputs of a selector.
3. Finally, check if the sets of control signals identified in the previous step are orthonormal. If the answer is yes, we have found a *complete* selector. If the answer is no, the control signals are only orthogonal and the selector is *incomplete*, but this can be fixed by adding another input to it.

1. BooleDozer is a trademark of IBM Corp.

Finding candidates

We start by looking for parts of the circuit that perform an *and–or* function. We do this by transforming the circuit to *nands* only, where *nand*–inverter–*nand* constructions without intermediate fanout are merged into a larger *nand*. Every two–level tree of *nands* now performs an *and–or* function, and is a *candidate* for (partial) replacement with a selector. The ideal form of this transformation, where the entire candidate can be replaced, is pictured in Figure 3. We will call the *nands* in the leftmost, input part of a

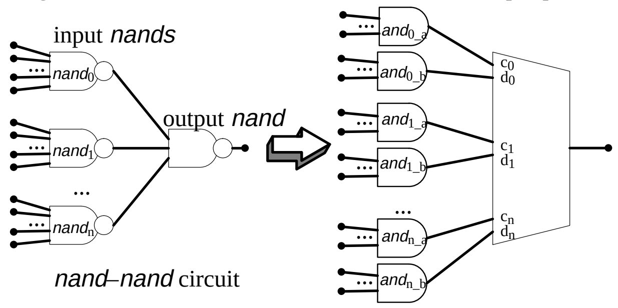


Figure 3: From *nand* tree to selector circuit

two–level *nand* tree the *input nands*. The *nand* generating the output is called the *output nand*. For every input *nand*, we have to find a partition of its input signals into a control and a data subset. The logical *and* of the control subset will form one control input of the selector, and the logical *and* of the data subset the data input. Another version of this transformation is when we only replace a part of the *n* input *nands* by a selector. In that case we have to keep a smaller output *nand* around, and invert the output of the selector (or use an inverting selector). This is pictured in Figure 4.

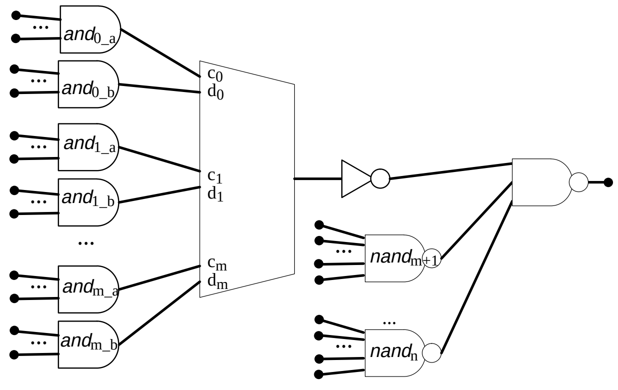


Figure 4: partial replacement of *nand* tree by selector circuit

Finding sets of control signals in candidates

The task we face is to find parts of two–level *nand–nand* candidates which can be implemented correctly by a selector. For each such selector we have to figure out which of the input signals to the input *nands* of a candidate form the control part of this select bit, and which signals the data part. We answer this question first with respect to *pairs* of input *nands* of candidates. For each such pair we are going to find *all* correct sets of signals which could be used as the control signal if these two *nands* were absorbed into a selector. The data signals are of course implicitly the remaining input *nand* inputs. If no data input remains, the constant value 1 is used. The sets of input signals which can be used as control signals for an input *nand* pair are called *matches* and form a *match matrix* for every candidate. Matches are formally introduced by the following definition:

Definition 9: A match m of a pair $(nand_i,\ nand_j)$ of input *nands* of a single candidate is a pair of non–empty sets of signals:

$$m = \{S_i,\ S_j\},\ S_i \subseteq Fanin(nand_i),\ S_j \subseteq Fanin(nand_j)$$
$$S_i \neq \emptyset,\ S_j \neq \emptyset,\ All\_1(S_i \cup S_j) = 0$$

Theorem 1: For a match $m = \{S_i,\ S_j\}$, let $c_1 = \bigwedge_{s \in S_i} s$, the logical *and* of the signals $s \in S_i$, and $c_2 = \bigwedge_{s \in S_j} s$, the logical *and* of the signals $s \in S_j$, then c_1 and c_2 are orthogonal signals.

Proof: We first prove the (normal) case when $S_i \not\subseteq S_j$ and $S_j \not\subseteq S_i$. Because of symmetry we only need to prove that $c_1 = 1 \Rightarrow c_2 = 0$. From $c_1 = 1$ it follows that $\exists_{v \in \mathcal{V}} \left(\bigwedge_{s \in S_i} s = 1 \right)$. Because there is a match $\{S_i,\ S_j\}$, it follows that $\forall_{v \in \mathcal{V}} \left(\bigwedge_{s \in S_i \cup S_j} s = \bigwedge_{s \in S_i} s \wedge \bigwedge_{s \in S_j} s = 0 \right)$. This can only be true if $\forall_{v \in \mathcal{V}} \left(c_1 = 1 \Rightarrow \bigwedge_{s \in S_j} s = 0 \right)$, from which $c_1 = 1 \Rightarrow c_2 = 0$ follows directly.

In the less likely case that $S_i \subseteq S_j$ or $S_j \subseteq S_i$, we get the following. Again, because of symmetry we only treat one case, $S_i \supseteq S_j$. $\{S_i,\ S_j\}$ is a match, so $\forall_{v \in \mathcal{V}} \left(\bigwedge_{s \in S_i \cup S_j} s = 0 \right)$. Also, $S_i \supseteq S_j \Rightarrow S_j \setminus S_i = \emptyset$. We can now write $\forall_{v \in \mathcal{V}} \left(\bigwedge_{s \in S_i \cup S_j} s = \bigwedge_{s \in S_i} s \wedge \bigwedge_{s \in S_j \setminus S_i} s = \bigwedge_{s \in S_i} s \wedge 1 = 0 \right)$.

Therefore, $\forall_{v \in \mathcal{V}}\ c_1 = 0$. A pair of signals, of which one is always 0, is orthogonal.

■

If we think of the inputs of an input *nand* as being ordered, we can represent a match between an input *nand* with n inputs and an input *nand* with m inputs as a pair of two bit vectors of length n and m respectively, where a 1 in the bit vector corresponds to an input signal being part of the match, and a 0 to a choice as data signal. An example of a match matrix is given in Figure 5. All matches between input *nand* 1 and input *nand* 2 are listed in match matrix entry (2, 1). Of course, the entry in (1, 2) would be identical, we don't need to store it. The diagonal does not carry any values either. But we do have to store $n/2 \times (n-1)$ entries if the output *nand* has a fanin of n.

Theorem 2: If $m = \{S_i,\ S_j\}$ is a match, then

$m' = \{S'_i,\ S'_j\},\ Fanin(nand_i) \supseteq S'_i \supseteq S_i,$ $Fanin(nand_j) \supseteq S'_j \supseteq S_j$ is a match too. If $S'_i \supset S_i \vee S'_j \supset S_j$, we call these matches *implied matches*.

Proof: For m holds: $All\_1(S_i \cup S_j) = 0$. This implies $\forall_{v \in \mathcal{V}} \left(\bigwedge_{s \in S_i \cup S_j} s = 0 \right)$. We need to proof that

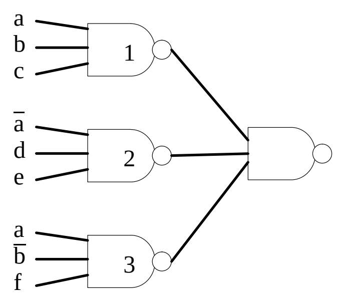
$$\forall_{v \in \mathcal{V}} \left(\bigwedge_{s \in S'_i \cup S'_j} s = 0 \right).$$

We know that $S'_i \cup S'_j \supseteq S_i \cup S_j$. We will define the difference set $S_{diff} = S'_i \cup S'_j \setminus (S_i \cup S_j)$. Now it is easy to see that

$$\forall_{v \in \mathcal{V}} \left(\bigwedge_{s \in S_i \cup S_j \cup S_{diff}} s = \bigwedge_{s \in S_i \cup S_j} s \wedge \bigwedge_{s \in S_{diff}} s = 0 \wedge \bigwedge_{s \in S_{diff}} s \right)$$

■

All matches with a gray background in Figure 5 are implied



	1	2	3
1		100 100 110 100 101 100 111 100 100 110 100 101 100 111 110 110 110 111 101 110 101 111 111 110 110 101 101 101 111 101 111 110 111 111	010 010
2			100 100
3			

Figure 5: Match matrix

matches. We only included them for entry (2, 1), for the other entries they can be constructed in a similar way. This observation leads to the following two rules:

1. When looking for matches, start looking for matches with as few signals as possible. This way we will find all matches that could not possibly be implied by another match first, and can easily skip testing of implied matches.
2. Store only the matches with minimal number of signals. All matches that are implied from this one can easily be derived and need not be stored.

There are two other ways we speed up the process of finding matches. The first property we exploit is given by the following theorem, which is in a sense the dual of Theorem 2;

Theorem 3: If $\{S_i, S_j\}$ is *not* a match, then $\{S'_i, S'_j\}$, $\emptyset \subset S'_i \subseteq S_i$, $\emptyset \subset S'_j \subseteq S_j$ is a not a match either.

Proof: If $\{S_i, S_j\}$ is not a match, it follows that $\exists_{v \in \mathcal{V}} \left(\bigwedge_{s \in S_i \cup S_j} s = 1 \right) \Rightarrow \exists_{v \in \mathcal{V}} \left(\forall_{s \in S_i \cup S_j} s = 1 \right)$. Therefore, for all $\{S'_i, S'_j\}$, $\emptyset \subset S'_i \subseteq S_i$, $\emptyset \subset S'_j \subseteq S_j$ holds $\exists_{v \in \mathcal{V}} \left(\bigwedge_{s \in S'_i \cup S'_j} s = 1 \right)$, and it is not a match.

■

From Theorem 3 it immediately follows that:

Corollary 4: For any two input $nands$ $nand_i$, $nand_j$ of a single candidate, if $\{Fanin(nand_i), Fanin(nand_j)\}$ is not a match, then there are no matches for this input $nand$ pair.

Thus, the first test we do for any input nand pair is to ask the oracle the value of $All\_1(Fanin(nand_i) \cup Fanin(nand_j))$. If the answer is 1, there are no matches for this pair. This single question to the Boolean Oracle can avoid many other questions to identify the matches.

The second thing we do is identifying counter examples for matches by performing random simulations. In each candidates *nomatch matrix* we store *nomatches*.

Definition 10: A nomatch nom of a pair $(nand_i, nand_j)$ of input $nands$ of a single candidate is a pair of non–empty sets of signals:

$$nom = \{S_i, S_j\},\ S_i \subseteq Fanin(nand_i)$$
$$S_j \subseteq Fanin(nand_j),\ S_i \neq \emptyset,\ S_j \neq \emptyset All\_1(S_i \cup S_j) = 1$$

During random simulation of the circuit after identifying the candidates, we observe the input pins of input $nands$ of candidates and store the nomatches. With this set of nomatches and Theorem 3 we can identify many pairs $\{S_i, S_j\}$ as definitely not being matches, and we do not have to bother the oracle for the decision.

We can now write down the algorithm to find the matches:

Algorithm 1: find matches

```
 1  for each candidate {
 2   for each pair (nand1, nand2) {
 3    if(All_1(nand1, nand2))
 4     continue; /* to next pair */
 5    for each match value m {
       /* lowest numbers of 1s first */
 6     if(m in nomatch matrix)
 7      continue;
       /* to next match value */
 8     if(m is implied match)
 9      continue;
        /* to next match value */
10     if(Oracle says All_1(m) = 0)
11      add match to match matrix;
12    }
13   }
14  }
```

The run time of algorithm 1 is quadratic in the number of input $nands$ of a candidate. If the test on line 3 fails, for a pair with a n

and a *m* input *nand*, we actually must check all $(2^n - 1) \times (2^m - 1)$ different possible match values. This is a very bad worst case time complexity, but in practical situations this algorithm turns out be fast enough even for large circuits. In our implementation the designer can limit the fanin of output *nands* and input *nands* of candidates to be considered to limit the run time if needed. By default, we do not consider candidates with more than 50 input *nands*, nor input *nands* with more than 20 inputs.

Table 1 shows how efficiently the heuristics in algorithm 1 limit the number of times we have to ask the Boolean Oracle if a match is valid. We have chosen the largest 10 circuits (in terms of number of connections in the circuit) from the MCNC combinatorial multi–level benchmark circuits in which we find selectors (see the Results section and Table 2) to show this. The size of a circuit is not directly related to run time for our algorithms, but this subset contains the 2 examples consuming the most CPU time. Column 'total tests performed' list the number of calls to the Boolean Oracle. Column 'total tests skipped' lists the number of times we decided we knew the answer before asking the oracle. Column 'initial pair test skip' shows the number of calls to the oracle we did not make because the test on line 3 of Algorithm 1 succeeded. Column 'nomatch skip' shows the number of times the test on line 6 succeeded. Column 'implied match skip' shows the number of times the test on line 8 succeeded, indicating a match was implied by a previously found match. This number is also equal to the number of matches that need not be stored. The actual number of matches we did have to store is listed in column 'matches stored'. The number of nomatches we found and stored during random simulation of the circuit with 1024 input vectors is listed in column 'nomatches stored'. The last column lists the CPU time in seconds needed on an IBM RS6000/370 (approximately 60 MIPS) to run the match finding algorithm. Table 1 shows that we succeeded in finding all matches, while making actual calls to the Boolean Oracle only in a small fraction of the cases. The number of matches stored for each example indicates the theoretical lower limit for the number of calls to the Boolean Oracle that need to be made. A real match can only be found by an All_1 test, as all shortcuts we use lead to a negative conclusion. All three tests in algorithm 1 contribute significantly to this result. The fact that we have chosen not to store implied matches also limited our storage needs for matches to a very acceptable number.

Table 1: Efficiency of match finding

name	total tests done	total tests skip	initial pair test skip	no–match skip	implied match skip	match es stored	nomat ches stored	cpu (s)
C5310	1473	11184	84	8242	2858	692	1776	14
C7552	2418	31459	23004	5893	2562	554	1541	44
dalu	873	18794	363	9914	8472	383	1319	16
des	6154	800117	9524	523347	267246	3006	15749	124
frg2	6679	115089	6580	84948	23561	195	1699	40
i10	1527	2.3E6	2.3E6	10491	4711	446	1751	28
i8	2934	83168	2180	61029	19959	1084	5699	38
pair	1275	5464	3588	1560	316	279	1996	21
x1	74543	2.6E9	2.6E9	3.7E6	1.3E6	235	60707	447
x3	1274	28092	13332	11356	3404	245	1472	15

From matches to selectors

We now know if and how pairs of input *nands* could be part of a selector. But, of course, we want the selectors to absorb as many of the input *nands* of a candidate as possible. In many respects this is not a simple task. It would be best if we could replace the entire candidate by a single selector. But in many cases this is not possible. In general we can replace a candidate by a number of selectors and some of the original input *nands*, much like in Figure 4. In many cases, several different configurations will be possible, some of which could be superior in some respect to others. We currently use a heuristic algorithm to find a set of 'good' selectors for every candidate. It is based on the concept of *growing* a selector consisting of 2 or more input *nands* by adding one more input *nand* to it. A selector is represented for these purposes by a set of input *nands*, each with a known partition of inputs into control and data inputs. A trivial starting point is of course taking a pair of input *nands* for which a match, specifying the partitions can be found in the match matrix. We can try to grow this configuration by adding one more input *nand* to it. This is only possible if the new input *nand* has a non–empty set of matches with the *nands* already chosen. In that case we have to figure out how the control subsets for construction of the larger selector should be.

As an example. let's consider the circuit of Figure 5. Remember that the match matrix does not actually have the gray entries. If we start with pair *nand*$_0$ and *nand*$_1$, we get the following selector:

nand$_1$	*nand*$_2$
100	100

If we try to add the third *nand*, the match matrix tells us that we have the following choices for the control inputs:

nand$_1$	*nand*$_2$	*nand*$_3$
100	100	
	100	100
010		010

The second line is in accordance with our original choice of control bits for *nand*$_0$ and *nand*$_1$, but the third line seems to be a contradiction. However, we have to remember that all implied matches are real matches too. From Theorem 2 it follows that we can add inputs to a match and obtain an implied match. This means that we can just do a bitwise *or* on the bitvectors. This leads to the following 3 bit selector:

nand$_1$	*nand*$_2$	*nand*$_3$
110	100	110

Or, graphically, to the selector configuration in Figure 6

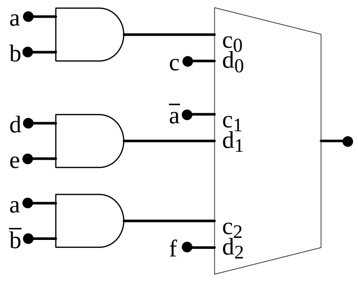


Figure 6: resulting selector circuit

From this we learn that we can add an input *nand* to a selector if the new input *nand* does have at least one match with all *nands* already in the selector. We say that the new *nand* is *compatible* with a selector if it has at least one match with all *nands* already in the selector.

To find all possible selectors for a candidate with *n* input *nands*, we would have to try to grow selectors towards all possible 2^n subsets of input *nands* of the candidate. This becomes too large a number very quickly, although it might be feasible for candidates of smaller size. We have chosen to use the following heuristic: we assume the input *nands* are ordered. We then try to grow selectors starting from each input *nand*, adding input *nands* later in the ordering if they are compatible. If they are not compatible, we try the next one. In this way, for each candidate, we obtain a subset of all possible selectors.

Checking completeness of selectors

In the previous discussions we have only made sure that the control signals for the selector are orthogonal. But we have not worried about the possibility of them not being orthonormal. Yet, this might lead to undesirable high impedance output values in some selector implementations. Therefore, we check this condition. For every selector of every candidate we temporarily create the *and* gates to get the control input values for this selector. The output nets of these *and*s are not connected. Let C be the set of control signals thus created. We then ask the Boolean Oracle what the value of $All\_0(C)$ is. If it is 1, the control inputs of this selector can be 0 at the same time, and the selector is incomplete. If we decide to implement this selector, we will have to make it complete. If $All\_0(C) = 0$, the selector is complete, and no further action is necessary.

We can make an incomplete selector complete by adding an extra bit to it. The data input of this bit is connected to the constant value 0, and the control bit is the logic *nor* of the other control bits, which effectively selects this bit when none of the other bits are selected.

As an example, the selector in Figure 6 is complete, because $a\ b + a\ \overline{b} + \overline{a} = 1$.

4 Merging selectors

In many cases it is possible to merge selectors found in different candidates into one larger selector. In fact, the ability to do this has proved important to construct large selectors. In many cases logic synthesis, or other restructuring, will change a circuit such, that the algorithms described in this paper only find small selectors initially.

One type of merging we can perform is when a selector output is connected (perhaps inverted) to the data input of another selector. We call this *input merging*. This situation is pictured for a simple example in Figure 7.

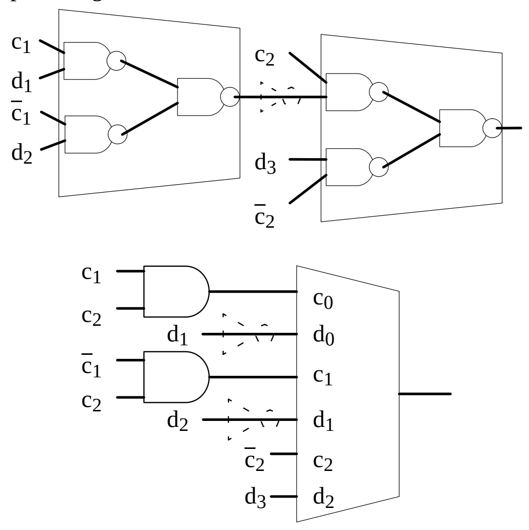


Figure 7: merging selectors: input merge

The second type of merging is pictured in Figure 8. In this case an input *nand*, which is part of a selector, is also an output *nand* of another selector. We call this, for obvious reasons, *halfway merging*. It is entirely equivalent to input merging with an inverter between the selectors.

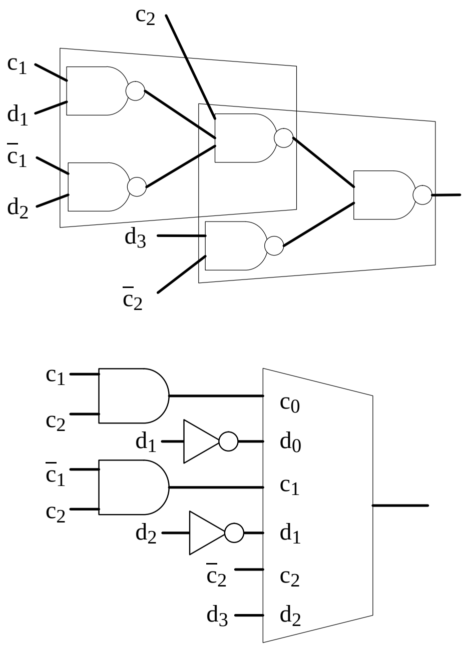


Figure 8: merging selectors: halfway merge

5 Results

The main results of this paper can be found in Table 2. It shows the selectors found in the entire set of MCNC multi–level combinatorial benchmark circuits (76 circuits). On the initial circuits we performed only some basic redundancy removal and a straight–forward conversion to *nands*. When we searched for selectors, we limited ourselves to candidates with at most 50 input *nands*. Each input *nand* was limited to 20 inputs. Both limitations assured us that we would have reasonable run times. The 6 circuits marked with ^{2}) hit these limits, and were not fully explored, but some selectors were found nonetheless. The 4 circuits marked with ^{1}) did run into the limits, and produced no selectors initially. On these examples we performed kernel factoring, after which they did not run into the limits any more and produced some selectors. In the benchmark circuits not listed in the table (C1355, C17, C432, C6288, cm138a, cm42a, decod, i3, i4, i5, majority, vda) we did not find any selectors.

Figure 9 summarizes the estimated saving in terms of the number of transistors for all benchmark circuits. It is obvious that quite a few circuits show considerable savings, up to 82%.

Table 2 lists all the selectors found in the circuits. Please note that there are no columns for 9 and 11 input selectors, because these were not found in this test.

The table column labeled ‘total sel bits’ gives the sum of all selector bits in the previous columns, to give an indication of the total selector contribution. In a pass transistor implementation this would be the total number of transistors used in all selectors. The column ‘trans. before 100%’ gives an indication of how many transistors the circuit used before the selectors were found. Because the circuits are transformed to *nands* only, this number is twice the total number of inputs to *nands*. This assumes a one–to–one mapping of *nand* expressions to *nands* from the library, which is of course not entirely accurate, but it is a good indication. The ‘100%’ indi-

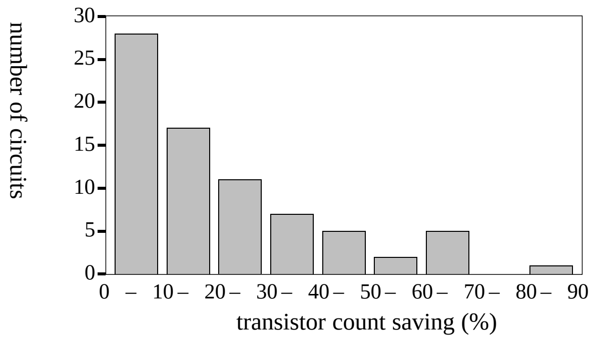


Figure 9: transistor count savings over all MCNC benchmark examples

cates that this number is used as a reference for other values. Column 'tr. after %' gives the percentage of transistors in *nands* left after putting in the selectors and their surrounding logic (*ands* to *and* the data and control bits, *nors* to generate the control signal for the extra selector bits added to make incomplete selectors complete). Clearly, in many cases a substantial amount of the original logic is replaced by selectors. Column 'tr. sel %' gives the number of transistors used in the selectors and the and–logic for the control and data bits. We assume this and–logic is also performed by serial pass transistors to the selectors. The quantity is expressed as a percentage of the original number of transistors. Column 'tr. *nors* %' gives the amount of transistors needed for the *nors*. Again we take twice the number of inputs to the *nors* here. This amount too is expressed as a percentage of the original transistor count. It is an indication of how much extra logic was necessary to make incomplete selectors complete. As you can see, in most cases this number is very small. Column 'total %', adds the previous three columns, to show the total % of transistors left after replacing *nands* by selectors.

Figure 10 summarizes the number of selectors of different sizes

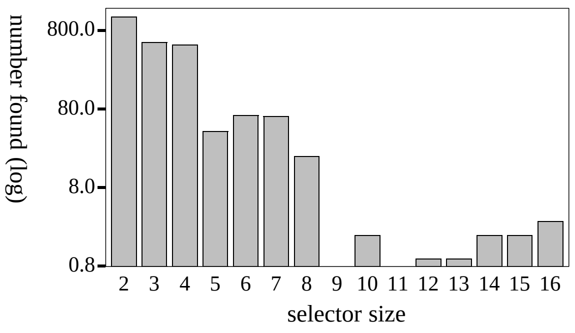


Figure 10: number of selectors of a specific size found in all MCNC benchmark examples

found in all benchmark examples combined. It clearly shows that many selectors were found, most of which were 8 bits wide or less. The final column of table 2 shows how much CPU time (in seconds on an IBM RS6000/370, a 60 MIPS machine) was spent to find and construct the selectors.

6 Conclusions

The theory and algorithms presented in this paper allow automatic mapping to selectors based on Boolean properties of signals in the circuit. Although the algorithms have an exponential worst case run time complexity, for practical circuits they are fast enough. We have shown that there is a large potential to area savings when pass transistor selectors would be used in the MCNC benchmark circuits.

Acknowledgements

The authors want to thank Dan Brand, Robert Damiano and Louise Trevillyan for many discussions and helpful hints during the implementation of the algorithms described in the paper in the IBM BooleDozer logic synthesis system.

References

[DET87] DETJENS, E, G. GANNOT, R. RUDELL, A. SANGIOVANNI–VINCENTELLI and A. WANG, "Technology Mapping in MIS", *Proceedings of the International Conference on Computer–Aided Design 1987*, pp. 116–119.

[KUN90]: KUNDA, R.P., P. NARAIN, J.A. ABRAHAM and B.D. RATHI, "Speed up of Test Generation using High Level Primitives", *Proceedings of the 27th Design Automation Conference*, June 1990, pp. 594–599.

[KUN92]: KUNDU, S., L.H. HUISMAN, I. NAIR, V.S. IYENGAR and L.N. REDDY, "A small Test Generator for Large Designs", *Proceedings of the International Test Conference*, Sept. 1992, pp. 30–40.

[LAN93]: LAN, S.H., R. GOPISETTY and K.R. DHARMARAJAN, "CMAP – Technology Mapping for Multiplexor Based FPGA Architectures Using Certificate of Generic Boolean Function", *Proceedings of the IEEE 1993 Custom Integrated Circuits Conference*, pp 3.4.1 – 3.4.4.

[MAI90] MAILHOT, F. and G. DE MICHELI, "Technology Mapping using Boolean Matching and Don't Care Sets", *Proceedings of the European Design Automation Conference 1990*, pp. 212–216.

[RUD93]: RUDELL, R., "Dynamic Variable Ordering for Ordered Binary Decision Diagrams", *Workshop Notes of the International Workshop on Logic Synthesis 1993*, May 1993, pp 3a–1 – 3a–12

[WHI83]: WHITAKER, S., "Pass–transistor networks optimize n–MOS logic", *Electronics*, September 22, 1983, pp. 144–148.

[ZHU93]: ZHU, K. and D.F. WONG, "Fast Boolean Matching for Field–Programmable Gate Arrays", *Proceedings of the 1993 European Design Automation Conference (EuroDAC)*, pp 352–357.

Table 2: Results on all MCNC multi–level benchmark circuits.

name	number of selectors of size													total sel bits	trans. before 100%	tr. after %	tr. sel %	tr. nors %	total %	CPU (s)
	2	3	4	5	6	7	8	10	12	13	14	15	16							
9symml	14	2												31	750	77	6	1	84	2.6
C1908	38						1							84	1802	68	7	1	76	6.2
C2670 [2])	29	2	25											164	2440	56	8	0.2	76	14
C3540	17	17	13	1	6	4	4				2	1	1	297	3948	63	11	1	75	27
C499	48													96	1856	69	5	0	74	4.4
C5315	77	15	78	3										526	6482	45	10	0.5	56	23
C7552	158	2	23	8										454	8266	66	7	0.9	74	45
C880	11	8												46	1336	88	13	3	94	4.6
alu2 [2])	2	3		3	1	1	2	1	1					79	1360	61	12	8	81	17
alu4 [2])	2	4	1	3	5	3		1		1		1	1	140	2726	61	12	8	81	50
apex6	15	67		5	11									322	2674	39	14	5	58	8.9
apex7	7	10				1								51	902	71	9	5	85	3.1
b1	2													4	46	57	9	0	66	0.6
b9	4													8	500	91	2	0	93	1.6
c8	15	7	1											55	710	43	17	3	63	3.7
cc		1												3	312	97	1	1	99	1.1
cht	35	1												73	678	16	20	0	36	2.8
cm150a													1	16	198	7	26	0	33	1.3
cm151a							1							8	100	32	20	0	52	0.9
cm152a							1							8	86	7	28	0	35	1.2
cm162a	4													8	174	82	5	0	87	1.0
cm163a	4													8	172	81	5	0	86	0.9
cm82a	2													4	100	76	4	0	80	0.6
cm85a	4													8	198	80	4	0	84	1.0
cmb	3													6	168	79	6	0	85	0.9
comp			1											4	368	95	2	1	98	1.4
cordic	8													16	358	75	4	0	79	1.7
count		16												48	544	47	15	0	62	2.2
cu	1		1											6	224	84	4	0	88	1.0
dalu	69	47												279	4278	61	9	3	73	17
des [2])	143	67	36	4	11	55	10							1182	15144	35	12	3	50	124
example2	21	7	8	1										100	1098	52	11	5	68	4.7
f51m	2	1	4											23	788	77	7	0	84	13
frg1 [1])	1													2	556	98	0.4	0	98	1.9
frg2	50	79	1											341	4622	60	13	1	74	40
i10	152	27	9	10	1	1								484	8218	66	8	2	76	28
i1		2												6	168	85	5	5	95	0.9
i2 [2])	1													2	876	99	0.3	0	99	3.4
i6	1		66											266	1752	22	20	0.3	43	7.7
i7	2		64											260	2188	19	15	0	34	11
i8		23	64	1	32									522	5538	45	11	0.5	57	39
i9		8	56											248	1954	33	13	0.7	47	9.1
k2 [1])	1	16	2				1							62	3104	89	3	3	95	8.5
lal	1	1												5	612	95	1	1	97	1.8
mux													1	16	198	7	26	0	33	1.5
my_adder			16											64	930	48	12	0	60	3.8
pair	95	44	2											330	5812	70	7	0.6	78	22
parity	1													2	240	95	1	0	96	0.8
pcle		1	7											31	238	30	19	19	68	1.3
pcler8		1	7											31	302	56	15	15	86	1.4
pm1		1												3	246	92	3	2	97	1.0
rot	18	14	2	2										96	3372	80	6	1	87	84
sct	3													6	502	90	3	0	93	1.6
tcon	8													16	98	2	16	0	18	0.9
term1	5	10	3											52	1422	83	7	3	93	5.5
t481 [1])	1	1	1	1										14	294	76	7	6	89	2.6
too_large [1])	3	8												30	2402	94	2	1	97	6.4
ttt2	17	5												49	1130	71	10	2	83	4.7
unreg	32													64	456	44	21	0	65	2.1
x1 [2])	8	2												22	4326	95	2	0.2	97	447
x2	1	1												5	166	70	11	0	81	0.9
x3	46	32	16											252	4386	73	8	2	83	15
x4	9	13	20											137	2020	58	10	2	70	8.1
z4ml		2	1											10	628	87	4	0	91	13
total	1191	568	528	42	67	65	20	2	1	1	2	2	3	7572					75	

[1]) No selectors initially, only after kernel factoring

[2]) Not fully explored , fanin limit exceeded