MULTI-LEVEL SYNTHESIS ON PALs

Gabrielle SAUCIER, Pascal SICARD
Institut National Polytechnique de Grenoble - Laboratoire CSI
46 avenue Felix Viallet
38031 GRENOBLE Cedex, FRANCE
Tel: 76574687, FAX: 76503421

Laurent BOUCHET
Laboratoire d'Annecy-le-vieux de Physique des Particules
Chemin de Bellevue BP 110
74941 ANNECY-LE-VIEUX Cedex, FRANCE
Tel: 50233324 poste 232, FAX: 50279495

ABSTRACT
This paper presents a decomposition method for logic mapping on PALs. It addresses a large variety of PALs. For functions having too many variables, a novel approach based on algebraic divisions is proposed. It is followed by an interesting reinjection phase tending to suppress useless decompositions. Both critical path and PAL numbers are optimised.

INTRODUCTION
Programmable Logic Devices (PLDs) are more and more successfully used to implement either combinational or sequential logic on electronics boards [13]. Several types of PLDs are presently available: the Programmable Read Only Memory (PROM), the Programmable Logic-base Sequencer devices (PLS) and the Programmable Array Logic devices (PALs). Some of them are often used to prototype ASICs and allow effective and low cost breadboarding. Recently, silicon compilers companies and dedicated companies have put a lot of efforts on logic synthesis but do not address specifically the PLDs target. Some other companies, like HP, Mentor graphics, announced dedicated softwares for logic synthesis on PALs but few details are available on the mapping methods used.

In the research area, only few papers are available on the topics [14,9]. A restricted class of PALs is considered in order to make the problem tractable (existence of bypassable output registers, pins which can be used as input or output). The method used is based on a progressive filling of the available PALs. The initial functions which do not fit in a PAL are partitionned in an initial step in subfunctions which will. The first function is assigned to a PAL. To fill the remaining place (if any), the best candidate among non assigned functions is chosen. The best candidate is the one having the most I/O in common and the fewest that are different. The process iterates with the next function. Such a procedure may be compared to a progressive placement method. More recent works [6] attempt to improve the above mentionned approach. Their main emphasis is put on an initial partitioning witch aimed at grouping product terms in clusters containing the same variables. Although different partitioning strategies had been developed, none of them bring actual progress compare to non partitioning strategies.

The system describes in this paper is somewhat different. First, an effort has been made to provide a language describing in an adequate manner PALs available at a given instant with the characteristics required to perform successfully the mapping. Written presently in Prolog, the efficiency has been proved on the MMI PAL library [MMI 88]. Extension to other families (ALTERA, INTEL, LATTICE) is under development. Second, it takes profit of the progresses in the synthesis area: two-level minimisation [3,5,7,10] as well as multi-level minimisation [1,2,4]. The decomposition problem is performed by "interesting" algebraic divisions. An algebraic division is interesting if the kernel, the cokernel and the remainder are easier to implement than the initial functions. This tends to control the number of levels of the final circuit. The selection criterion is the number of variables in the kernel and in the remainder. After decomposition, an original reinjection phase aims at suppressing unnecessary decompositions to optimize the decomposition. The tool presented in this paper is part of the ASYL system which is a complete system dedicated to the synthesis of digital systems [12].

This paper is organized as follows. In section 1, the synthesis problem on PALs is introduced. Sections 2, 3 and 4 will discuss in details the three steps of the synthesis algorithm. Section 5 reports on the results on a set of benchmark examples. A conclusion and future work orientation will be presented in the last section.
1. Formulation of the problem

A PAL is a PLA where the OR plane is fixed and where each product terms in the AND plane may be used once, and only once, in the OR plane.

PAL devices can be characterized first by the number of inputs (8 to 20 in the case of the MMI family), the number of outputs (1 to 10) and the number of product terms per output (2 to 16). Inputs are provided in true and complement forms. PAL devices can also be classified according to the following characteristics: fixed or programmable output polarity (in the case of a positive (resp. negative) output polarity, the output is the sum (resp. the complement of the sum) of the product terms programmed in the AND plane); internal feedbacks (these internal connections between outputs and inputs are usually coupled with programmable three-state gates); the number and the type of internal flip-flops used to provide registered outputs. These different characteristics can be mixed altogether: some PAL devices exist with registered outputs, programmable polarity and feedback to one input.

From a set of boolean functions and a data base (library of PALs), the ASYL system produced a netlist of PALs and jdec fusemap. The functions are described as a sum of product terms together with some informations relevant to their implementations (registered or combinational output, name of the register control pins, ...). Each PAL device in the library is described by its function and its structure (number of I/O pins, type of polarity, ...).

Starting from these specifications, the system selects a minimal set of devices and give an implementation of the functions minimizing the number of levels. A cost will be associated with each device (e.g. the number of pins) allowing to evaluate the cost of a given implementation. The algorithm used to implement a set of functions is made up of three successive steps that will be further discussed:

(1) two-level local minimization of both the functions and their complements. A local minimisation is performed to reduce the number of product terms;
(2) decomposition of each function which does not fit on any PAL of the library, in subfunctions that will;
(3) final assignment of the functions and subfunctions to a minimal set of PALs of the library.

2. Complementation and local minimization

Some PAL devices have a programmable output polarity and therefore we can choose between the implementation of the function or of its complement. Then, the complement of each function is computed. Once the direct (F) and the complemented (~F) forms of each function are obtained, we realize a local minimization of F and ~F. Recall here that a product term in a PAL is used by only one output.

3. Local decomposition and reinjection phase

3.1. Local decomposition

Some of the functions obtained at this stage cannot be directly implemented on the available devices. A function that cannot be directly implemented on a PAL of the library will be rewritten in terms of subfunctions. This will obviously increase the number of levels and therefore the propagation delay associated with the function. As the propagation delay is a critical parameter of the implementations, we rewrite the functions while minimizing the number of additional levels. This can be the case for the 4 following reasons:

Case 1: Only the complement of a function can be implemented. Let \( F = \sum m_i \) and \( ^{\sim}F = \sum m_j \), we just have to introduce a new function \( F' \) which is the complement of \( F \) and to rewrite as follows: \( \begin{align*} F &= F', \quad ^{\sim}F = {^{\sim}F}' \end{align*} \)

Case 2: A registered function cannot be directly implemented. In the same way as previously, we just rewrite \( F = F', \quad ^{\sim}F = {^{\sim}F}' \) where \( F' \) equal \( F \) without register and \( ^{\sim}F' = \sum m_i, \quad ^{\sim}F' = \sum m_j \). In a first step we realise \( F' \) and in a second step \( F \).

Case 3: Function has too many variables. In this case, there is no device in the library with enough input variables to implement either the function or its complement. The decomposition process is based on algebraic division and kernel extraction [2]. A kernel algebraic division \( F = K.Q + R \) is interesting because:
- the number of variables in the kernel \( K \) and in the quotient \( Q \) is inferior to the number of variables in the function \( F \);
- the number of product terms in the kernel \( K \), in the quotient \( Q \) and in the remainder \( R \) is inferior to the number of product terms in the function \( F \);
- the number of variables in the remainder \( R \) is not always inferior to the number of variables in the function \( F \) but it decrease with the number of product terms.

The algorithm is the following:

I use, between the function and its complement, the
function with the minimal number of product terms, let \( F = \Sigma m_i \) be this function;
II) compute all the kernels of \( F \), then 2 case are considered :

**Case a**: if there is a kernel different of \( F \) then it is possible to factorize \( F \). Choose the best kernel \( k \) and rewrite \( F = q.k + r \) where \( q \) is the quotient \((\Sigma F/k)\) and \( r \) the remainder of the algebraic division of \( F \) by \( k \). A kernel will be interesting if the associated quotient, remainder and the kernel itself are easy to implement, i.e if their implementations require a minimal number of further decompositions. If so, the number of recursive decompositions will be limited and therefore the number of levels in the final circuit. With regards to the propagation delay, the length of the longest path in the decomposition tree has to be reduced, producing a well-balanced decomposition tree. The selected kernel has to minimize simultaneously the number of variables it contains and the number of variables in \( r \). So the chosen kernel verify: \( \text{MAX} \{ \text{number of variables in } k, \text{number of variables in } r \} \) is minimal. The decomposition process is iterated until \( q, k \) and \( r \) can be implemented;

**Case b**: if there is no kernel different from \( F \) then there is no immediate factorization of \( F \). In this case, each variable in \( F \) appears at most twice: once in its normal form and once in its complemented form. The algorithm used in this case will not be discussed in details. Let us mention that two main subcases appeared: first, \( F \) is made up of a single product term, then the algorithm splits it into a set of subfunctions with a balanced number of variables in each subfunction; second, \( F \) is made up of more than one product term, then the algorithm will distinguish if all the variables in \( F \) are unate or double [15].

Considering the following example illustrating the case 3:

\[
F = v_1.v_2.v_3.v_4.v_5.v_6 + v_1.v_2.v_4.v_6.v_7.v_8.v_9.v_{10}
\]

The kernels are:

\[
v_4.v_5.v_1.v_2.v_4.v_7.v_8.v_9.v_{10}
\]

\[
v_3 + v_4.v_6
\]

the function itself, \( F \).

Kernel \( v_3 + v_4.v_6 \) is selected as it minimizes at the same time the number of variables it contains and the number of variables in the remainder. Therefore \( F \) is rewritten as:

\[
F = k.q + r \quad \text{where:} \quad k = v_3 + v_4.v_6; \quad q = v_1.v_2; \quad r = v_4.v_5.v_6.v_7.v_8.v_9.v_{10}
\]

**Case 4**: Function has too many product terms. In some cases, neither the function nor its complement can be directly implemented on a PAL of the library owing to a too large number of product terms. But if there is a device with enough inputs, it is possible to rewrite this function using a single additional level. Moreover, in the final assignment step, this decomposition will allow an optimum use of the most adequate PALS and therefore minimize the total number of devices. The algorithm is the following:

I) choose, between the function and its complement, the function with the minimal number of product terms, let \( F = \Sigma m_i \) be this function;
II) select a device \( D \) with enough inputs to implement \( F \) and with a maximal number of product terms (non registered outputs);
III) split the set \( \{ m_i \} \) into balanced subsets and rewrite \( F = \Sigma S_{i_j} \) and \( \Sigma F = \Pi S_{i_j} \)

such the subfunctions \( S_{i_j} \) can be implemented on device \( D \). The criterion to select the device \( D \) tends to limit both the number of additional subfunctions and the number of required devices. The number of levels is 2 as in a first step the subfunctions \( S_{i_j} \) are evaluated in parallel and in a second step \( F \) is realised.

### 3.2. Re-injection phase

According to the previous algorithm, we will obtain a decomposition tree (figure 1), but some of these decompositions may be useless. Once the decomposition is made, we will try to eliminate some subfunctions by re-injecting them in the upper level of the decomposition. As the efficiency of this re-injection is order-dependent, we need a heuristic to determine a near-optimal list of successive re-injections. In the decomposition tree, we will first try to re-inject subfunctions in the longest path of the tree and then in the other paths. As the propagation delay depends on the depth of the tree, this heuristic also minimizes the propagation delay in the final result. Consider the decomposition tree for a given function \( F \) given in figure 1, figures between brackets are respectively the number of variables and of product terms of the current function. Suppose also that the PALS used have a maximum number of 10 input variables and of 6 product terms.

**Figure 1**: Decomposition and re-injection tree

<table>
<thead>
<tr>
<th>Number of Variables</th>
<th>Number of Product Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

544
The initial decomposition leads to the decomposition given in figure 2a (4 levels and 8 subfunctions). Reinjections 1, 2, 3 and 4 lead to the decomposition given in figure 2b (2 levels and 4 subfunctions).

**4. Assignment to device**

At this stage of the algorithm, all the subfunctions can be assigned to a device of the library. As either the subfunctions or their complements can be in most of the cases mapped on the devices, the assignment step will consist, for each function, in selecting the form of the function to be implemented. The goal of this step is to minimize the total number of devices. The first difficulty is that each function can be assigned to a large set of devices, and the second one is that each device may usually be used to implement several functions. The algorithm (figure 3) tries all the possible assignments for each function, trying to use already existing devices.

Once an initial solution has been computed, other solutions are searched by backtracking on the successive assignments. The backtracking in the search space is limited by a branch and bound heuristic. The tree of the possible assignments has to be reduced in order to reach a near-optimal solution in reasonable time. The first point is that we will first assign the functions that can be assigned to a minimal number of devices. The second point is that the functions will be assigned to the devices with a maximal number of outputs, thus trying to get a maximal number of functions on the same device.

Before assigning the functions to devices, the algorithm firstclassifies both the functions and the devices. Let \( #F_i \) (resp. \( #F_i \)) the number of devices to which the function \( F_i \) (resp. \( F_i \)) can be assigned.

The functions are classified according to the following ordered criteria:
- \( #F_i \) increasing,
- minimal of the number of product terms of the function and of its complement decreasing,
- number of variables decreasing.

For each function, the direct and complemented forms \( F_i, \neg F_i \) will be ordered according to the two following ordered criteria:
- \( \#F_i \) increasing,
- number of product terms in \( \neg F_i \) decreasing.

As usually the number of variables is the same in \( F_i \) and \( \neg F_i \), we will first try the form which can be assigned on the maximum number of devices. In this way, we have an increased probability to use an already selected device. The devices are classified according to the two following ordered criteria:
- number of outputs decreasing,
- number of variables decreasing,
- number of product terms per output decreasing,
- programmable polarity.

The same classification will be applied when selecting a device from the list of already selected devices during the assignment step.

---

**figure 3 : Mapping algorithm**
5. Practical results

The presented algorithms have been implemented in Prolog (compiled version of DELPHIA) on SUN workstations under UNIX, on a VAX under VMS and on a CETIA workstation under UNIX. The library used is made up of 32 PALs from MMI: pall10h8, pall12h6, pall16h2, pall16c1, pall10i8, pall12i6, pall16i8, pall16r4, pall16i2, pall16r8, pall16r6, pall16p8, pall16r6p8, pall16r6p4, pall12i10, pall14i8, pall16i6, pall18i4, pall20i2, pall20c1, pall20i10, pall20i8, pall20r6, pall20r4, pall20s10, pall20rs4, pall20rs8, pall20rs10. And it could be easily extended to other family like ALTERA, INTEL, LATTICE, as the description language is flexible enough. The algorithms have been validated on a set of both combinational and sequential examples which are standard benchmarks. Computation times are indicated on a SUN 3/50 workstation under UNIX.

It doesn't matter to prove, in the first time, the efficiency of PLDs integration compared with the SSI; everybody knows that the integration ratio is from 6:1 to 8:1 [11,9]. But it will be very interesting to compare these results (table 1) with others synthesis on PALs tools.

Note that for some examples, like vg2 in table 2, the number of devices from the whole library is greater than the number of devices from the reduced library (initial library without pall16h2, pall16c1, pall16i2, pall16r8, pall20c1, pall20r8, pall20r6, pall20r4). It is due to the fact that some devices in the whole library (e.g. pall20c1, pall16c1) have a large number of variables and product terms for a single function. This facility is not often used in the reduced library. But if the reduced library decreases the number of devices, it increases the number of levels and therefore the propagation delay (respectively 2 levels using the whole library as compared with 5 levels using the reduced library for vg2).

Conclusion and future work

The results show that ASYL-PAL implements in an effective way boolean functions on a set of PALs minimizing both the number of PALs and the critical path. A connection to the controller synthesis tool of ASYL allows also to start from a controller specification. The computation times are low even in the Prolog prototype. A C-version speeds up the present results.

References