Abstract: This paper presents an approach to high-level synthesis which is based upon a 0/1 integer programming model. In contrast to other approaches, this model allows solving all three subtasks of high-level synthesis (scheduling, allocation and binding) simultaneously. As a result, designs which are optimal with respect to the cost function are generated. The model is able to exploit large component libraries with multi-functional units and complex components such as multiplier-accumulators. Furthermore, the model is capable of handling mixed speeds and chaining in its general form.

1 Introduction

During the recent years, there has been an ever-increasing demand to speed up the design cycles for the design of electronic systems. This demand is caused by time-to-market requirements for products in this area. At the same time, there has been an increasing need to achieve for correctness by construction.

Due to these driving forces, synthesis techniques are now being used for the design of many electronic products. Currently, logic synthesis and synthesis of finite state machines are the most widely employed techniques. Unfortunately, these techniques cannot be used for the design of efficient systems containing data paths.

The design of such systems is the goal of the various approaches to high-level-synthesis. Despite significant efforts by researchers in the area, high-level synthesis is still hardly used in industry. A major reason for this is the inefficiency of current high-level synthesis systems. This inefficiency has several reasons. We believe, the two most important reasons are

1. the partitioning of algorithms for high-level synthesis into algorithms for solving the three subtasks (scheduling, allocation and binding) independently. However, these subtasks are related and therefore this approach potentially results in inferior designs.

2. library mapping — in contrast to its popularity for logic synthesis — is still very poor.

In our paper, we will describe a method which potentially improves the efficiency of high-level synthesis significantly by avoiding these two sources of inefficiencies. The synthesis system based on our method is called OSCAR (optimum simultaneous scheduling, allocation and resource binding). OSCAR also respects timing constraints — a feature that was missing in early synthesis systems.

2 Related work

Almost all early approaches to high-level synthesis partitioned the problem into subproblems for scheduling, allocation and binding. This includes the work by Tseng and Siewiorek [13], Marwedel [10] and others (see [11] for a survey). This work helped finding solution methods for these subproblems. Later it was recognized, that these subproblems should be solved simultaneously in order to avoid suboptimal results.

The work of Gebotys is especially stimulating in this respect, because it is based on a formal integer programming (IP)-model, which has the potential of solving several subproblems concurrently. The approach to scheduling in this work is an improvement over an earlier model by Hwang [7]. Early work by Gebotys into this direction [4] did not include binding. Recently, the two-index model in [4] has been extended into a three index model (see e.g. [5]). The three-index model has the potential of handling advanced features such as pipelining, mixed speeds and wiring optimization. Unfortunately, the latter is not described in [5].

Other approaches which are based on IP-models include the following: Library mapping (see also Dutt [3]) is emphasized in a paper by Achatz [1], integrating scheduling and allocation. Allocation and assignment have been integrated for example by Rim and Jain [12]. In addition to having the potential for solving subproblems concurrently, IP-models have the advantage of being formal models of high-level synthesis. This makes formally checking the correctness of high-level synthesis easier.

Now that a considerable amount of heuristics have been published, we believe that is it the time to investigate more formal models. Currently available IP-models, however, do still have major limitations. For example, currently published IP-models do not allow chaining in its general form. For all existing algorithms, operations to be chained must be manually replaced by a single, more powerful operation before actual synthesis is started. This means, these algorithms cannot automatically decide whether or not two operations are to be chained.

Moreover, existing algorithms usually consider simple libraries containing mostly adders and multipliers. They are not capable of exploiting efficient complex components such as multiplier-accumulators and many of them are unable of selecting components with different speeds. Our work aims at removing the limitations mentioned above.

1Reprint from latex source. Copyright of the original publication: IEEE Computer Society.
3.1 Preprocessing

The input description of our high-level-synthesis system OSCAR is based on a subset of VHDL [8]. The input description is first compiled into a special form of data-flow graphs, namely Gajski’s assignment decision diagrams (ADD) [14, 6]. These diagrams have been extended to include timing specifications. These extended ADDs are called timed assignment decision diagrams (TADDs).

3.2 Integrated Scheduling, Allocation and Binding

3.2.1 Binding model

High-level synthesis basically has to establish bindings between operations \( j \), control steps \( i \) and resources \( k \). Such bindings can be represented by binary decision variables. Triple-indexed variables are required for integrating scheduling, allocation and binding.

\[
x_{i,j,k} = \begin{cases} 
1, & \text{if op. } j \text{ will be started on inst. } k \text{ at cs } i \\
0, & \text{otherwise}
\end{cases}
\]

Throughout this text, we will use index \( k \) as a name for a particular component. \( K \) will be the index set (value range) of these names.

Each component \( k \) will be an instance of a component type contained in the component library. We will use index \( m \) to denote a certain component type and \( M \) to denote the set of names of all component types. Function \( type \) is assumed to return the component type of a certain component instance.

Furthermore, we will use index \( j \) to uniquely denote an operation contained in the TADD. \( J \) will be the set of all \( j \)‘s. More precisely, each \( j \) corresponds to an operation instance. Each \( j \) is of a certain type, e.g., a particular operation in the TADD may be an instance of operation type \( \text{add} \) or \( \text{multiply} \). We will use \( g \) to denote a certain operation type and \( G \) to denote the set of all operation types. Function \( optype(j) \) is assumed to return the operation type of a certain operation instance \( j \).

Table 1 contains the used mathematical symbols.

<table>
<thead>
<tr>
<th>Variable/Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_{i,j,k} )</td>
<td>( i ), ( j ), ( k ) variables to be computed by the synthesis system. Will assign a control step ( i ) and a component instance ( k ) to each operation instance ( j ). All combinations of ( i ), ( j ) and ( k ) for which no solution is feasible will not be used as subscripts of ( x ). For example, if ( k ) cannot perform operation ( j ), the corresponding decision variables are never generated in order to reduce the number of variables and relations.</td>
</tr>
<tr>
<td>( x_{i,j,k} )</td>
<td>( i ), ( j ), ( k ) variables to be computed by the synthesis system. Will assign a control step ( i ) and a component instance ( k ) to each operation instance ( j ). All combinations of ( i ), ( j ) and ( k ) for which no solution is feasible will not be used as subscripts of ( x ). For example, if ( k ) cannot perform operation ( j ), the corresponding decision variables are never generated in order to reduce the number of variables and relations.</td>
</tr>
<tr>
<td>( x_{i,j,k} )</td>
<td>( i ), ( j ), ( k ) variables to be computed by the synthesis system. Will assign a control step ( i ) and a component instance ( k ) to each operation instance ( j ). All combinations of ( i ), ( j ) and ( k ) for which no solution is feasible will not be used as subscripts of ( x ). For example, if ( k ) cannot perform operation ( j ), the corresponding decision variables are never generated in order to reduce the number of variables and relations.</td>
</tr>
</tbody>
</table>

3.2.2 Constraints

The following constraints are required to define the set of legal solutions:

**Table 1: Mathematical notation**

1. **Operation assignment constraints**

Each component type \( m \) is assumed to be able to execute a set \( G_m \) of operation types. E.g., a certain component type may be able to perform additions and multiplications while others are only able to perform either of the two.

Our first set of constraints now models the fact that each operation \( j \) should be started on exactly one resource instance of the appropriate type. Furthermore, each operation \( j \) should be started in a control step \( i \) which lies within the range \( R(j) \) of feasible control steps. \( R(j) \) is the range of control steps between the earliest (ASAP) and latest (ALAP) control step feasible for operation \( j \). These conditions are modelled by the following relations:

\[
\forall j \in J : \sum_{i \in R(j)} \sum_{k \text{ executable on } i} x_{i,j,k} = 1 \tag{1}
\]

For each \( j \), the sum over \( k \) includes only instances for which the relation “\( j \) executable on \( k \)” holds. This ensures that operations will be mapped to appropriate components. Relation “\( j \) executable on \( k \)” can be defined as:

\[
j \text{ exec. on } k \iff optype(j) \in G_{type(k)} \tag{2}
\]

Note that relation “\( j \) executable on \( k \)” is more general than the corresponding implicit relation in [4]. For each \( k \), we have to know the corresponding type \( m \) before solving our synthesis problem. To model this knowledge, we are using a function called \( type \). For each potential instance \( k \), function \( type \) has to return the corresponding resource type. Without loss of generality, we require \( type \) to be a monotone step function of \( k \). Before synthesis, a sufficiently large number of potential instances is automatically computed for each type \( m \). In order to model the fact that for a certain
A simple observation can be used to speed up the search for optimal designs: if type(\(k\)) = type(\(k+1\)), then the solutions \(b_k = 1, b_{k+1} = 0\) and \(b_k = 0, b_{k+1} = 1\) are equivalent, except for renaming of resource instances. In order to generate only one of these equivalent solutions, we require that

\[
\forall k : \text{ if } \text{type}(k) = \text{type}(k+1) \text{ then } b_k \leq b_{k+1} \quad (3)
\]

without loss of generality.

Experimental results have shown that this constraint can reduce the execution time of the IP-solver by a factor of 10 - 30! This redundancy is not eliminated in other models [5].

Components which we have considered so far are able to perform certain functions. Results computed by these functions can be described in terms of expressions involving operators, input ports and constants. Up to now, high-level synthesis systems have only considered expressions involving a single operator, e.g. \(i_{\text{IN}} a + i_{\text{IN}} b\). In the following, we will call components computing these expressions simple components.

Recent component libraries, however, do contain components such as multiplier-accumulators (MACs), which compute expressions like \((i_{\text{IN}} a * i_{\text{IN}} b) + i_{\text{IN}} c\). These components correspond to complex gates in logic synthesis and we will therefore call these components complex components. Complex components allow very efficient implementations, but high-level synthesis systems in general are not capable of exploiting them. One of the goals we set for OSCAR is to decide automatically whether to map sets of adjacent operations to several simple components or to a single complex component. To this end, we define macro-operations\(^2\) to be a set of adjacent operations which can be executed by at least one component type. Let \(Y\) be the set of all such macro-operations.

Constraint (4) ensures that either all operations contained in a macro-operation \(y \in Y\) are assigned to separate simple components or to a single complex component. \(\forall j \in J:\n\sum \sum_{x_{i,j,k} \in R(j)} x_{i,j,k} + \sum \sum_{y \in Y : y \in R(j)} x_{i,y,k} = 1 \quad (4)\)

If no complex components are employed, the right sum of (4) becomes zero. In this case the constraint reduces to the standard operation assignment constraint (1).

Additionally, constraint (5) restricts the assignment of macro-operations to at most one complex component:

\[
\forall y \in Y : \sum \sum_{y \in R(j)} x_{i,y,k} \leq 1 \quad (5)
\]

\(^2\)Note that macro-operations \(y \in Y\) can be handled in the following constraints just as conventional operations \(j \in J\).
4. Chaining Constraints

In case that two or more data-dependent operations are assigned to the same control step the sum of real execution times must be less than the user defined cycle time \(\text{time}_{\text{cycle}}\). We define a new relation

\[ j_1 \not\preceq j_2 \iff j_1 \prec j_2 \land \exists k_1 : j_1 \text{ executable on } k_1, \exists k_2 \neq k_1 : j_2 \text{ executable on } k_2 \]

\[ \text{time}_{\text{cycle}} \geq \text{time}(j_1, k_1) + \text{time}(j_2, k_2) + \ell_{\text{phys}} \]

Constant \(\ell_{\text{phys}}\) describes a system dependent latency caused by interconnect delays. Further, we define a new set \(\text{CHAINS}\). Each \(ch \in \text{CHAINS}\) is the longest chain \(j_1 \prec \cdots \prec j_n\) (1 \(\leq \) \(n\) are local indices) in the data flow graph with:

1.) operations \(j_1\) to \(j_n\) are data dependent:
\[ \forall j_i, i \in \{1, \ldots, n - 1\} : j_i \prec j_{i+1} \]

2.) at least two operations \(j_i, j_{i+1} \in ch\) can be executed in the same control step: \(j_i \not\preceq j_{i+1}\)

3.) \(ch\) is maximal:
\[ \bigwedge j_0 : j_0 \not\preceq j_1 \land \bigwedge j_0 : j_0 \not\preceq j_{n+1} \]

\[
\text{CHAINS} := \bigcup ch \quad \text{with} \quad \forall j_i, i \in \{1, \ldots, n - 1\} : j_i \in J \land \forall j_i, i \in \{1, \ldots, n - 1\} : j_i \not\preceq j_{i+1}\]

Figure 1 illustrates the effect of chaining guided by a simple expression tree. A cycle time of 100\(\text{ns}\) and latency of 10\(\text{ns}\) restrict the total execution time of all chained operations to at most 90\(\text{ns}\). The left side of the figure represents two possible solutions with maximum number of chained operations in one control step.

In this example, \(ch_1\) consists of \(\{j_1, j_3, j_4\}\) because of \(j_1 \prec j_3 \prec j_4\) and \(j_1 \not\preceq j_3, j_3 \not\preceq j_4\). The other set \(ch_2\) is represented by \(\{j_2, j_3, j_4\}\). Set \(\text{CHAINS}\) consists of \(\{ch_1, ch_2\}\).

In case of solution I, constraint (8) is fulfilled only for the first two elements of \(ch_1\) and \(ch_2\). The alternative solution shown underneath consists of the entire set \(ch_1\). The additional assignment of operation \(j_2 \in ch_2\) in the same control step would violate constraint (8).

5. Timing constraints

Models for timing constraints can be taken over from [4]. For example, if two operations \(j_1\) and \(j_2\) should be separated by \(T\) control steps, then the following relations should hold:

\[ \forall i : i \in R(j_1) \]

\[ \sum_{i \in R(j_1)} x_{i, j_1, k} + \sum_{i \in R(j_2)} x_{i, j_2, k} \leq 1 \]

Minimum timing constraints\(\text{\(^{\text{3}}\)}\) can be represented by the following relation, respectively:

\[ \forall i : \sum_{i \in R(j_1)} x_{i, j_1, k} + \sum_{i \in R(j_2)} x_{i, j_2, k} \leq 1 \]

6. Register constraints

Register constraints can also be taken over from [5].

3.2.3 Cost function

One of the objectives of the current paper is to show how interconnect optimization can be integrated into a unified model for scheduling, allocation and binding. Therefore, the cost function has to include terms describing interconnect. The following cost function fulfills this requirement:

\[ \sum_{m \in M} (c_m \cdot \sum_{i \in E, \ell_{\text{phys}}(i) = m} b_k) + \sum_{k_1, k_2} c_{k_1, k_2} \cdot w_{k_1, k_2} \]

\(^{3}\)in order to formulate maximum timing constraints exchange the \(\preceq\) resp. \(\lnot\preceq\) relations of the sum boundaries
The first term describes the cost of functional units, \( c_m \) is the cost per instance of component type \( m \).
The second term describes the cost of the interconnect, \( c_{k_1, k_2} \) is the cost for interconnecting \( k_1 \) to \( k_2 \). Due to the lack of layout information, \( c_{k_1, k_2} \) is usually set to \( 0 \) bitwidth.

A naïve implementation for computing \( w_{k_1, k_2} \) would contain terms of the form \( x_{i,j,k_1} \times x_{i,j,k_2} \), which are quadratic in \( x \). The corresponding quadratic assignment problem can be avoided by using a trick published by Rim, Jain and De Leone [12]. The trick consists in defining \( w_{k_1, k_2} \) as: \( \forall j_1 < j_2 \):

\[
\begin{align*}
    w_{k_1, k_2} &\geq \left( \sum_{i_1 \in R[j_1]} x_{i_1,j_1,k_1} + \sum_{i_2 \in R[j_2]} x_{i_2,j_2,k_2} \right) - 1 \quad (13) \\
    w_{k_1, k_2} &\geq 0 \quad (14)
\end{align*}
\]

If both variables \( x_{i,j_1,k_1} \) and \( x_{i,j_2,k_2} \) are set to 1, \( w_{k_1, k_2} \) also becomes 1. In the case that only one of both variables takes the value of 1, \( w_{k_1, k_2} \) becomes 0. Constraint (14) avoids negative values if both variables are set to 0. Consider that a variable \( w_{k_1, k_2} \) is only created if both \( j_1 \) and \( j_2 \) can be executed on instances \( k_1 \) resp. \( k_2 \). With this trick, the cost function is still a linear function in \( x \).

Hence, algorithms for solving (linear) integer programming problems can be used to compute optimal bindings.

4 Results

We have applied our synthesis system to several benchmarks. All calculated results are optimal. The execution times have been measured on a Sparc 10 using the mixed IP-solver [2].

In the following, we present experimental results for the 5th-order Elliptical Wave Filter [9] and the Differential Equation Solver benchmark.

5th-Order Elliptical Wave Filter
Table 2 shows the results of the EWF employing adders, multipliers and multi-functional units with different delays, latencies and costs. All delays are measured in control steps. The entry 2:1 means a pipelined function unit with a delay of two cycles. The specified costs represent the size relations between the particular component types.

Table 3 presents results with chained (left part) and unchained (right part) operations. Chaining was applied to + and * operations which allows to perform both operations together within one control step. Note, that all component delays are specified in ns. For this example, the application of chaining yields a reduction by 3 control steps of the minimum schedule presuming a clock cycle of 1000ns. Table 4 presents synthesis results based on a complex component library. We employed MACs which are able to perform the addition and multiplication (separately or as macro-operation) within one control step.

Table 2: Effect of different speeds of components

<table>
<thead>
<tr>
<th>( \text{FU} )</th>
<th>( + )</th>
<th>( * )</th>
<th>( (+,*)</th>
<th>( + )</th>
<th>( * )</th>
<th>( (+,*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>delay</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>costs</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>cs 1</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3: Enabling vs. disabling chaining of add/mult-operations

Differential Equation Solver
The results calculated for the Diff-Eq benchmark are given in the tables 5-7.

Table 7 presents synthesis results of interconnect minimization. The number of interconnects could be reduced without increasing the number of required components.

5 Conclusion

We presented a new IP-model and its implementation in OSCAR which extends existing approaches to IP-based high-level synthesis into six different directions:

1.) OSCAR extends previous models for scheduling and allocation to wiring optimization.
2.) In addition, OSCAR is able to handle complex libraries with multi-functional components.
3.) Furthermore, OSCAR is able to handle components with mixed speeds.
4.) OSCAR has the potential to exploit the existence of complex components such as multiplier-accumulators.
5.) OSCAR has the capability of assigning several operations operating on short bit vectors to components with a large bit width.
6.) Finally, OSCAR allows chaining in its general form.
We have shown that acceptable runtimes can be achieved for standard benchmarks.

References


Table 4: Employing complex component libraries

<table>
<thead>
<tr>
<th>FU</th>
<th>delay</th>
<th>costs</th>
<th>cstep</th>
<th>delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>cs-3</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>cs-4</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>cs-5</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>cs-6</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>cs-7</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5: Enabling vs. disabling chaining of sub/operations

Table 6: Effect of different speeds of components

Table 7: Enabling vs. disabling interconnect minimization