TOWARDS GLOBAL SCHEDULING AND REGISTER ALLOCATION USING PREDICATED EXECUTION

Rogério Xavier de Azambuja — *UFSC[1] – ULBRA Santa Maria[2]*
Felipe Vieira Klein — *UFSC[1]*
Flávio Meurer — *UFSC[1]*
Luiz C. V. dos Santos — *UFSC[1]*

[1]UFSC – Universidade Federal de Santa Catarina
INE/CTC/LAPS – P.O. Box 476 – 88010-970 – Florianópolis/SC, Brazil
{xavier, fvklein, flavio, santos}@inf.ufsc.br

[2]ULBRA Santa Maria – Universidade Luterana do Brasil
P.O. Box 21834 – 97020-001 – Santa Maria /RS – Brazil

ABSTRACT

This paper presents an approach for register allocation and scheduling which relies on three main ideas: global optimization, solution space exploration and on-the-fly generation of a symbolic state machine. To allow global optimizations while preserving semantics, the traditional notion of control dependence is replaced by the notion of predicate. In our approach, predicates are used not only as attributes of operations during scheduling (predicated execution), but also as attributes of values during register allocation. Experimental results show that this global approach improves the chances of reaching high-quality solutions in High-Level Synthesis.

KEYWORDS

High-Level Synthesis, Scheduling, Register Allocation, Conditional Execution.

RESUMO

Este artigo apresenta uma abordagem para o escalonamento e a alocação de registradores fundamentada em três idéias principais: otimização global, exploração do espaço de soluções e geração simultânea da maquina de estados finitos. Para permitir otimizações globais preservando a semântica, a tradicional noção de dependência de controle é substituída pela noção de predicado. Em nossa abordagem, predicados são usados não apenas como atributos de operações durante o escalonamento (“predicated execution”), mas também como atributos dos valores durante a alocação de registradores. Os resultados experimentais mostram que esta abordagem global aumenta a possibilidade de alcançar soluções de alta qualidade em Síntese de Alto Nível.

PALAVRAS-CHAVE

Síntese de Alto Nível, Escalonamento, Alocação de registradores, Execução condicional.

TOWARDS GLOBAL SCHEDULING AND REGISTER ALLOCATION USING PREDICATED EXECUTION

Rogério Xavier de Azambuja UFSC[1] – ULBRA Santa Maria[2]
Felipe Vieira Klein UFSC[1]
Flávio Meurer UFSC[1]
Luiz C. V. dos Santos UFSC[1]

[1]UFSC – Universidade Federal de Santa Catarina
INE/CTC/LAPS – P.O. Box 476 – 88010-970 – Florianópolis/SC, Brazil
{xavier, fvklein, flavio, santos}@inf.ufsc.br

[2]ULBRA Santa Maria – Universidade Luterana do Brasil
P.O. Box 21834 – 97020-001 – Santa Maria /RS – Brazil

ABSTRACT

This paper presents an approach for register allocation and scheduling which relies on three main ideas: global optimization, solution space exploration and on-the-fly generation of a symbolic state machine. To allow global optimizations while preserving semantics, the traditional notion of control dependence is replaced by the notion of predicate. In our approach, predicates are used not only as attributes of operations during scheduling (predicated execution), but also as attributes of values during register allocation. Experimental results show that this global approach improves the chances of reaching high-quality solutions in High-Level Synthesis.

1. INTRODUCTION

High-Level Synthesis (HLS) tools generate the structure of an application-specific integrated circuit (ASIC) from the description of its behavior [3] [4]. The design of ASICs under tight time and resource constraints is challenging. Tight constraints ask for tools with growing optimization capabilities and demand design space exploration.

To face this challenge, we propose an approach which combines global optimization techniques and exploration of alternative solutions. When addressing scheduling and register allocation, optimizations are performed beyond basic blocks boundaries.

To allow such global optimizations while preserving semantics, the traditional notion of control dependence is replaced in our approach by the notion of predicate. The idea of using predicates as attributes of operations is well-known in the Computer Architecture domain [5], where predication is supported in hardware. Predicates have also been used in High-Level Synthesis [9][10] as a way of modeling conditional execution. In our approach, predicates are used not only as attributes of operations, thereby modeling conditional and speculative execution, but also as attributes of produced and consumed values so as to allow global register allocation.

The main distinguishing features of our approach are the following:

- Our modeling of conditional execution does not rely on the notion of control dependence, which limits the exploitation of parallelism, but on the notion of predicate.
- Our approach is based on exploration of alternative solutions, instead of applying a package of heuristic criteria to generate a single solution.
- Our scheduler associates operations directly with states, thereby building a state machine on-the-fly. As a result, not only the schedule latency, but also the number of states can be assessed as a way of evaluating the quality of a solution.
- A global modeling is proposed for register allocation under conditional execution. The notion of *life interval* [4], which assumes a linear sequence of scheduled operations, is replaced by the notion of a *path* in the state machine graph on which a produced value has to be preserved until its final consumption. This notion can be described by means of predicates attributed to the production and consumption of values.

This paper is organized as follows. Section 2 summarizes our modeling of design representation with respect to constraints and conditional execution. Our high-level synthesis approach is sketched in Section 3. Section 4 focuses on our techniques for global scheduling and register allocation. Section 5 addresses implementation and experimental results. Finally, our conclusions and perspectives are discussed in Section 6.

2. OUR MODELING

The main notions used throughout this paper are formalized in this section.

Definition 1 - A *data flow graph* DFG(V,E) is a directed graph where each vertex $v_i \in V$ represents an operation and each edge $(v_i, v_j) \in E$ represents a data dependence between v_i and v_j.

Definition 2 - A *state machine graph* SMG(S,T) is a directed graph where each vertex $s_i \in S$ represents a state and each edge $(t_i, t_j) \in T$ represents a transition between states s_i and s_j.

Definition 3 - A *resource constraint vector* **a** is a vector where each component a_t represents the number of available resources of a given type t.

Definition 4 - The *execution delay* d_i is the number of cycles needed to complete the execution of operation v_i.

Definition 5 – The *latency* λ of a given SMG is the number of states in the longest path from the source to the sink vertex.

Definition 6 - Given the states s_p, $s_k \in S$, the *distance* between them, written $\delta(s_p, s_k)$, number of edges in the path from s_p to s_k.

Definition 7 - Given an arbitrary state $s_k \in S$, the set of *available operations* in s_k, written A_k, is the set of all operations v_j, such that:

- Operation v_j was not scheduled in s_k neither in any state s_m reaching s_k.
- For each immediate predecessor v_i of v_j scheduled in some arbitrary state s_p, the inequality $\delta(s_p, s_k) \geq d_j$ holds.

Definition 9 – Given an arbitrary state $s_k \in S$, the set of *unfinished operations* in s_k, written U_k, is the set of all operations v_j, such that:

- Operation v_j was already scheduled in some arbitrary state s_m reaching s_k.
- The inequality $\delta(s_m, s_k) < d_j$ holds.

Definition 10 – Given two arbitrary states s_i and $s_j \in S$, s_i and s_j are *equivalent*, written $s_i \equiv s_j$, if and only if $A_i = A_j$ and $U_i = U_j$.

Definition 11 – Let s_i e $s_j \in S$ be two arbitrary states. Given a value v produced in state s_i and consumed for the last time in state s_j, the *life interval* of value v, written I_v, is the interval starting in the clock cycle associated with state s_i and finishing in the cycle associated with s_j.

Definition 12 - The *chromatic number* χ is the minimum number of colors in the solution of a vertex coloring problem [4].

Definition 13 - A *test* T_i is an operation whose result is a Boolean value. Each test is associated with a Boolean variable c_i, called a *guard*.

Definition 14 - The predicate $G(\psi)$ is a Boolean function defined on the set of guards $\{c_1, c_2, ..., c_n\}$, where ψ can be an operation v_i, an edge (v_i, v_j) or a state s_i as detailed below.

Let us clarify how conditional execution is modeled with an example. Figure 1a shows a behavioral description containing two nested conditional constructs, defined by tests T_1 and T_2. Figure 1b shows the corresponding DFG, as well as the predicates of each operation. Note that, since operations A, B, T_1 and H are executed inconditionally, to their predicates is assigned the Boolean constant 1. Note also that the operations which execute only when the result of test T_1 is true include the guard c_1 in their predicates. On the contrary, since operation G only executes when the outcome of T_1 is false, the complement of the guard, written $\overline{c_1}$, is included in its predicate. Finally, notice that operation E, which belongs to the inner conditional construct, executes when T_1 is true and T_2 is false. Therefore, its predicate is $c_1 \cdot \overline{c_2}$. To the predicates edges are attributed the corresponding predicates from its operation source, used to identify the end of the conditional test.

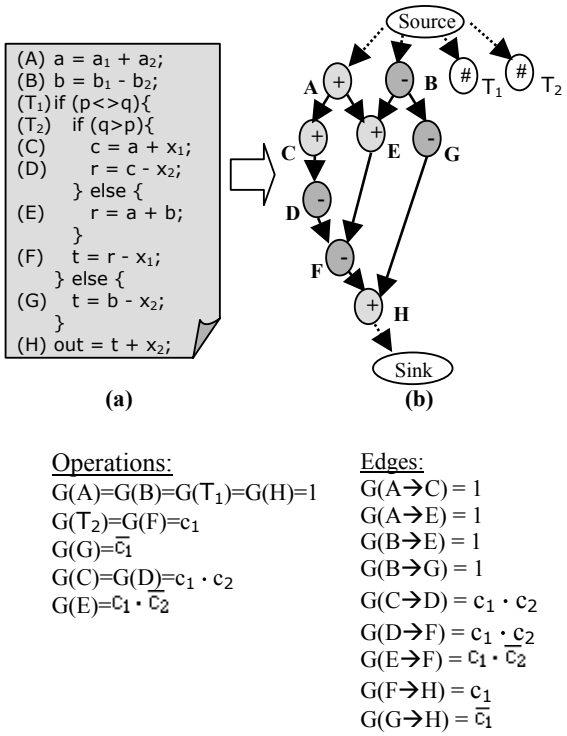


Figure 1: The modeling of conditional execution with predicates

3. RELATED WORK

Several methods have addressed behavioral descriptions containing conditional construct in High-Level Synthesis. For instance, Tree Scheduling [6] uses a tree structure to represent different control paths and allow the motion of operations. Path-based Scheduling [2] [3] produces a schedule for each control path independently and then unifies them into a single schedule. In the hierarchical approach described in [8], a DFG with conditional construct is transformed into an "equivalent" DFG without conditional constructs which is scheduled by a conventional basic-block scheduler. A condition vector is

proposed in [13] to model mutually exclusive operations and to support speculative execution. A condition vector is essentially a way of representing a predicate. Other approaches [9] [10] have generalized this notion later.

4. AN OVERVIEW OF THE APPROACH

To increase the chances of finding a good solution under tight constraints, it is mandatory not to restrict the search space to a single solution. A better approach would be to systematically generate several solutions, then evaluate their costs and finally select the best solution. We adopt the second approach, which is cast as the interaction of two main engines [10], an *explorer* and a *constructor*, as depicted in Figure 2.

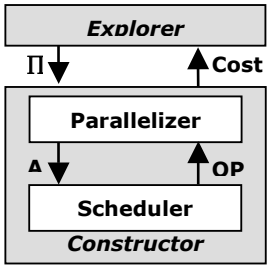


Figure 2: An overview of the approach

The task of the explorer is to create priority encodings for the operations to be scheduled and send them, one at a time, to the constructor. The constructor builds a solution for each priority encoding $\prod$ and return its cost to the explorer. The explorer receives the cost of a given solution, compares it with the costs of previous solutions and decides whether the new solution is satisfactory or not.

Different metrics can be used to define the cost of a solution. For instance, in this paper we use the latency λ (see Definition 5) as the cost for scheduling and the chromatic number χ (see Definition 12), as the cost for register allocation.

The constructor consists of a *parallelizer* and a *scheduler*. Given a new state s_k in the SMG, the main task of the parallelizer is to find the set of operations available for scheduling within the limited amount of resources in that state. The evaluation of the set A_k of available operations (see Definition 7) is the key to global scheduling. Since we have replaced control dependencies with predicates to model conditional execution, several precedence constraints are actually removed, uncovering more parallelism. As a consequence, the set A_k may contain operations coming from different basic blocks.

The task of the scheduler is simply to select as many operations from A_k as the number of available resources. If two operations map to a same resource, the priority encoding $\prod$ will break ties. Given a state s_k, the scheduler returns the set of operations scheduled in that state, written OP_k.

5. PREDICATION IN SCHEDULING AND REGISTER ALLOCATION

We have adopted the same structure of the scheduling algorithm proposed in [1] and adapted in [10] to fit the approach sketched in Section 2. However, the original algorithm assumes a control-flow graph (CFG) where control dependencies are represented. The original algorithm uses CFG-based data flow analysis to find available operations in each state. In our approach, since we rely on a DFG where only data dependencies are represented, conditional execution is captured by predicates. As a consequence, the evaluation of available operations involves checking both the DFG and operation predicates.

The behavior of our parallelizer is summarized in Algorithm 1.The procedure Paralellizer(**a**, $\prod$) creates a current state s_k and schedule in it as many operations as can be accommodated into the available resources.

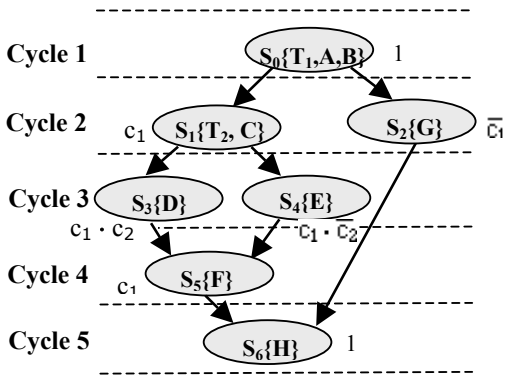
After all possible operations are scheduled in s_k, the procedure creates a next state s_{k+1}, which will become the current state in the next loop iteration. Therefore, operations are scheduled into successive states until all operations are scheduled. States to be scheduled are maintained in the list Next. The algorithm terminates when such list is empty.

It is important to notice that when a conditional test T_n is scheduled in state s_k, two next states are created: s_{k+1} (corresponding to guard c_n) and s_{k+2} (corresponding to guard $\overline{c_n}$). Note that the operations from A_k selected to become available at states s_{k+1} and s_{k+2} are subjected to complementary guards, i.e., conditional execution is taken into account.

```
Parallelizer (a, Π){
  create initial state s0 in the SMG;
  A0 = { vi |vo is the only predecessor of vi in the DFG};
  insert s0 in the list Next;
  while (Next ≠ ∅) do{
    sk = first element of the list Next;
    sj = FindEquivalentState(sk);
    if (sj ≠ none) {
      for each predecessor sn of sk {
        remove transition (sn, sk) ∈ T;
        insert edge (sn, sj) in T;
      }
    }
    else{
      OPk = ScheduleState(sk, Ak, Uk, a, Π);
      P = FindUnfinishedOperations(OPk);
      D = FindAvailableOperations(OPk);
      D = D ∪ Ak;
      if (∃ conditional test Tn ∈ OPk whose guard is cn) {
        create two new states sk+1 and sk+2 in the SMG;
        Ak+1 = SelectAvailableUnderGuard(Ak, cn);
        Ak+2 = SelectAvailableUnderGuard(Ak, c̄n);
        Uk+1 = Uk+2 = P;
        insert sk+1 and sk+2 in the list Next;
      }
      else{
        create one new state sk+1 in the SMG;
        Ak+1= D;
        Uk+1 = P;
        insert sk+1 in the list Next;
      }
    }
  }
}
```

Algorithm 1: Scheduling Algorithm

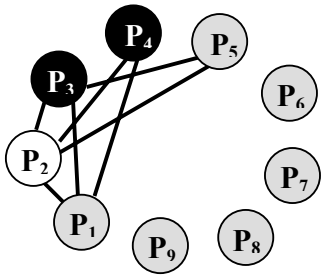
An example of scheduling is illustrated in Figure 3a. It shows a SMG obtained from the DFG in Figure 1a, assuming one resource of each type (adder, subtractor, comparator). Note, for instance, that operations D and E are both executed in cycle 3. However, their predicates tell us that their execution is mutually exclusive (G(D)•G(E)=0). In other words, they are scheduled on different SMG paths. Notice also that operation C executes speculatively, since it executes before the result of test T_1 is known. In a classical representation, C would be control dependent upon T_1, limiting speculative execution. This precedence constraint is removed in our modeling, uncovering more parallelism. The extra parallelism can be accommodated into idle resources, which is likely to reduce latency.



(a) SMG (λ=5 and |S|=7)

	Begin	End
P_1(A→C)	S_0	S_1
P_2(A→E)	S_0	S_4
P_3(B→E)	S_0	S_4
P_4(B→G)	S_0	S_2
P_5(C→D)	S_1	S_3
P_6(D→F)	S_3	S_5
P_7(E→F)	S_4	S_5
P_8(F→H)	S_5	S_6
P_9(G→H)	S_2	S_6

(b) Life paths



(c) Conflict Graph (χ=3)

Figure 3: An example on how predication is used

Register allocation would be trivial if a different register were allocated to each distinct value. However, the same register can be shared by values whose life intervals (see Definition 11) do not overlap [4]. This sharing policy assumes that values are produced by operations scheduled in a linear-time sequence. In other words, it implicitly assumes basic block scheduling. When conditional constructs are present in the behavioral description, scheduling actually produces a symbolic finite-state machine, represented by our SMG. If we preserve the notion of interval under conditional execution, either we preclude global register optimization or we overestimate the number of registers. Besides, under conditional execution it may happen that a same operation be scheduled in states belonging to different SMG paths. For those reasons, a generalization of the notion of life interval is needed, as formalized below:

Definition 15 – Let s_i e $s_j \in S$ be two arbitrary states of the SMG. Given a value v produced in state s_i and consumed for the last time in state s_j, the *life path* of v, written P_v, is the path starting at s_i and finishing at the immediate predecessor of s_j.

Figure 3b shows the life paths for all the values produced in the behavioral description in Figure 1a, whose SMG is shown in Figure 3a.

Since intervals are generalized into paths, intercepting intervals are generalized as intercepting paths. This is formalized in Algorithm 2. Given two values u and v, the procedure ExistConflict(u,v) detects whether or not their storage would be conflicting by checking if P_u intercepts P_v.

```
ExistConflict (u, v){
    if (Pu ∩ Pv ≠ ∅)
        return(TRUE);
    else
        return(FALSE);
}
```

Algorithm 2: Procedure for conflict detection between values.

Figure 3c shows a conflict graph where each vertex represents a life path. When building such a graph, Algorithm 2 decides whether or not there must be an edge between a pair of vertices. Notice that although there are nine life paths, only three registers are required to store their respective values (χ =3), as indicated by the vertex coloring shown in Figure 3c.

6. IMPLEMENTATION AND EXPERIMENTAL RESULTS

A prototype of our approach is implemented in C++ under the Linux operating system. Predicates are implemented using binary decision diagrams. We have used the CUDD package [11] to support predicate manipulation. Due to the frequent update of available and pending operations they are efficiently kept in binary heaps.

In our experiments, we have used three classical examples from the HLS literature. They are referred to as **Wakabayashi** [13], **Kim** [8] and **rotor** [9]. For each example, 1000 alternative solutions were generated and examined. For these examples, the best solution was always reached within the first 100 solutions. Experiments were performed under the same assumptions made in [9] and [10].

6.1. Scheduling Results

Tables 1, 2 report the latency λ obtained by other methods as compared to ours. The first columns show the resource constraints. Since our method generates the state machine on-the-fly, we are also able to report the number of states (|S|). The methods in [7] and [12] rely on heuristics to generate a single solution. The results of an exact method are reported in [9]. Although exact, this method guarantees the optimal solution only for a restricted speculative execution model [10]. Note that our approach reaches the best known latencies for all tested resource constraints.

Table 1: Latency for the example Kim.

ADD	SUB	COM	Our approach		λ in [9] Optimal Solution	λ in [7]	λ in [12]
			λ	|S|			
2	1	1	6	14	6	7	6

Table 2: Latency for the example Rotor.

ALU	MUL	TAB	COM	Our approach		λ in [9] Optimal Solution
				λ	|S|	
2	0	1	-	7	16	7
3	0	1	-	7	13	7
4	0	1	-	6	13	6
2	2	1	-	8	16	8
3	2	1	-	8	14	8
4	2	1	-	8	13	8

6.2. Results for Register Allocation

Tables 3 and 4 show our results for the solution with best latency. These results are compared with those obtained by other methods. It was difficult to find methods which address the problem under exactly the same conditions. This difficulty precludes the direct comparisons with other methods. In spite of that, we present the values we found as a reference. Typical numbers of registers obtained by conventional allocation are reported in [14], but their respective latency is not informed. In [15] a different problem is tackled, where the goal is to find a scheduling for a fixed number of registers and, again, latencies are not informed. Observe that for the example Wakabayashi, we have obtained the best reported values. For the example Kim, we have obtained a worse value than in [14]. However, since the respective latency is not reported, the better result found in [14] is likely to belong to a solution with larger latency than ours, which would explain the smaller number of registers. In [9], which describes a method also based on predicates, register allocation is unfortunately not addressed for examples **rotor** and **s2r**.

Table 3: Number of registers for the example Wakabayashi.

ADD	SUB	COM	Our approach		χ in [14] Typical allocation		χ in [14]		χ in [15]	
			λ	χ	λ	χ	λ	χ	λ	χ
2	1	1	7	4	-	5	-	4	-	4

Table 4: Number of registers for the example Kim.

ADD	SUB	COM	Our approach		χ in [14] Typical allocation		χ in [14]		χ in [15]	
			λ	χ	λ	χ	λ	χ	λ	χ
3	2	1	5	7	-	-	-	-	-	4
3	3	1	5	7	-	8	-	4	-	-

7. CONCLUSIONS AND PERSPECTIVES

This paper has presented a way of using predication to allow *global* scheduling and *global* register allocation. Experimental results for scheduling show that our modeling can reach the best known latencies for all the tested examples. Few results were published about register allocation under conditional execution. In spite of the ensuing difficulty to making direct comparisons, our experimental results seem promising. In any case, in the face of the scarcity of published results about global allocation in HLS, our results are already a contribution. Such scarcity is perhaps an evidence that this topic deserves further investigation and more experiments should be performed and reported.

8. REFERENCES

[1] AIKEN, A.; NICOLAU, A. "A Resource-Constrained Software Pipelining", IEEE Transactions on Parallel and Distributed Systems, vol. 6, nº 12, december of 1995.

[2] BERGAMASCHI, R. A. et al. "Control-Flow Versus Data-Flow Based Scheduling: Combining Both Approaches in an Adaptive Scheduling System", IEEE Transactions on VLSI Systems, vol. 5, nº 1, pp. 82-100, 1997.

[3] CAMPOSANO, R. "Path-based Scheduling for Syntesis" IEEE Trans. On Computer-Aided Design, vol 10, nº 1, january of 1991.

[4] DE MICHELI, Giovanni "Synthesis and Optimization of Digital Circuits", Mc Graw-Hill, USA, 1994.

[5] HENNESSY, J. L.; PATTERSON, D.A. "Computer Architecture: A Quantitative Approach", Morgan Kaufmann Publishers, Second Edition, 1996.

[6] HUANG, S. H. "A Tree-based Scheduling Algorithm for Control Dominated Circuits", 30th ACM/IEEE Design Automation Conference, pp. 578-582, 1993.

[7] KIM, T. et al. "A Scheduling Algorithm for Conditional Resource Sharing - A Hierarchical Reduction Approach", IEEE Trans. on CAD, vol 13, nº 4, april of 1994.

[8] KIM, T.; Liu JANE W. S.; Liu, C. L. "A Scheduling Algorithm For Conditional Resource Sharing", IEEE, USA, 1991.

[9] RADIVOJEVIC, I.; BREWER, F. "A New Symbolic Technique for Control Dependent Scheduling", IEEE Trans. on Computer-Aided Design, vol. 15, nº 1, p. 53 , 1996.

[10] SANTOS, Luiz C. V. dos "Exploiting instruction-level parallelism: a construtive approach", Eindhoven University of Technology, PhD. Thesis, Eindhoven, 1998.

[11] SOMENZI, Fábio "University of Colorado – USA(http://vlsi.colorado.edu/~fabio)", january of 2001.

[12] WAKABAYASHI, Kazutoshi; TANAKA, H. "Global Scheduling Independent of Control Dependencies Based on Condition Vectors", Proc. ACM/IEEE Design Automation Conference, pp.112-115, 1992.

[13] WAKABAYASHI, Kazutoshi; YOSHIMURA, Takeshi "A Resource Sharing and Control Synthesis Method for Conditional Branches" IEEE, Japan, 1989.

[14] ZHAO, Q.; EIJK, C. A. J. Van "Register Binding for DSP Code Containing Predicated Execution", IEEE, Eindhoven, 1999.

[15] ZHAO, Q.; EIJK, C. A. J. Van; PINTO, C. A. Alba; JESS, J. A. G. "Register Binding for Predicated Execution in DSP Applications", IEEE, Eindhoven, 2000.