Using Prolog and CLP for

Designing a Prolog Processor

Abstract

This paper presents the LPCAD (Logic Programming-based system for CAD) system for designing complex microprogrammed processors using Prolog and Constraint Logic Programming (CLP). The system is developed around VHDL, the industry standard language for hardware description and simulation, and has been used for designing a Prolog processor based on the Warren Abstract Machine (WAM). LPCAD accepts a VHDL high level specification of the instruction set and the datapath specification of the processor to be designed as inputs, and provides the microprogrammed control memory and a structural VHDL model of the whole processor as outputs. Benchmark programs can be applied to the structural model, extracting information from the simulation results used to motivate design decisions. Definite Clause Grammars (DCG's) are used to compile (parsing and code generation) the behavioral description of the processor's instruction set into a Register Transfer Level (RTL) code. CLP is used for optimal planning (resource assignment and scheduling) of the RTL operations resulting from compilation.

Alvaro Ruiz-Andino Illera
Dpto. Informática y Automática
Fac. CC. Matemáticas
Universidad Complutense de Madrid
Ciudad Universitaria
28040 Madrid, SPAIN
Tel.: (34) 1 - 3 94 44 68
Fáx: (34) 1 - 3 94 46 87
e-mail: arai@dia.ucm.es

José J. Ruz Ortiz
Dpto. Informática y Automática
Fac. CC. Físicas
Universidad Complutense de Madrid
Ciudad Universitaria
28040 Madrid, SPAIN
Tel.: (34) 1 - 3 94 43 75
Fáx: (34) 1 - 3 94 46 87
e-mail: jjruz@dia.ucm.es
1. Introduction

The design of a complex system like a WAM [2][3] based special purpose processor for Prolog execution requires a sound methodology in order to avoid mistakes and increase the overall performance of the processor. The design methodology introduced in this paper is based, on one hand, on VHDL [1] simulation capabilities, and on the other hand, on LPCAD, a tool which automatically synthesizes the control unit's micro programs, starting from the VHDL specification of the processor's instruction set, and also generates the processor's structural description in VHDL from a specification in Prolog of the datapath and a VHDL standard component library. We have chosen VHDL as the input specification language for two main reasons: first, because of its wide use for industrial hardware development, and second, because it allows simulation of the instruction set behavior specification, as well as the structural model of the datapath (along with the synthesized micro-programs). Functional verification of the instruction set is achieved through simulation of its VHDL specification, using a set of benchmarks Prolog programs [2] which are generally selected to stress specific aspects of Prolog implementations. Once the control unit's micro-programs are generated, the VHDL structural model of the processor is simulated, gathering performance measurements (resource usage statistics and execution times) using the same set of benchmarks. With this methodology, the final processor design is reached through successive refinements of the instruction set and datapath, guided by the performance measurements obtained through simulation.

Micro-program synthesis requires breaking down the processor's instruction set into simple RTL operations, assignment of datapath resources to each operation, and the scheduling of the operations. Resource assignment and scheduling must be done in such an optimal way that allows the maximum number of operations to be executed in parallel. LPCAD was implemented using Prolog and CLP. Prolog's formalism DCG's [4] greatly simplifies parsing the VHDL sources, syntactic analysis and RTL operation generation. Optimal resource assignment and scheduling implies a wide search space problem, which can be easily stated in a declarative way using CLP [5] over Finite Domains [7], without losing significant efficiency in comparation with an imperative implementation.

The organization of this paper is as follows: Section 2 includes a brief introduction to VHDL hardware description language, an overview of the design methodology and environment, and presents the way processor is specified. Section 3 discusses parsing, analysis, and RTL operations generation. Section 4 explains the preliminary resource assignment step, and Section 5 presents the solution given to the optimal resource assignment and scheduling problem, using CLP. Finally, Section 6 discusses achievements, conclusions and some ideas for future work.

2. Processor's Design Methodology and Environment.

Hardware description language VHDL was developed with the support of the US DoD in order to gain uniformity in the description of design specifications. Later on it has become IEEE standard 1076-1987, revised in 1992. The design of the language was driven by simulation aspects, which is a main application of the language. A VHDL program is a set of concurrently running processes intercommunicated via signals. The behavior of each process is described by means of a sequential algorithm making use of imperative constructions similar to those in ADA: variable assignment, if, case, while statements, etc. Besides, the language provides two special statements for process
intercommunication: wait statement and concurrent signal assignment. VHDL also allows hierarchical descriptions in order to describe an architecture by means of component instantiation and their interconnections (structural specification).

2.1. Design Environment.
Figure 1 shows the overall organization of the design environment. This design environment allows easy and fast evaluation of different instructions sets running on different architectures. Refinements of the instruction set and datapath specifications are guided by the performance measurements gathered in the simulation of the structural model. The designer describes the architecture of the processor (datapath and instruction set) to be analyzed by simulation. The datapath is built with components chosen from a specific library. The VHDL structural description generator reads the datapath specification in Prolog and generates a VHDL structural program. Prolog facts directly map onto a set of component instantiation statements corresponding to design units stored in the library. The micro-program generator synthesizes the control operations needed to implement the processor instructions onto the specified datapath. Synthesis involves three main steps: 1) parsing and analysis of the VHDL source of instruction set behavioral description and its translation into simple register transfer level operations, 2) a preliminary resource assignment to each operation, and finally 3) optimal scheduling and resource assignment.

![Figure 1 Design Environment](image-url)
2.2. High Level Specification.
The instruction set of the processor to be designed is specified using a single VHDL process describing the behavior of all instructions and auxiliary routines. Within this specification memory areas and registers are declared as local variables of different types, and there is no reference to any structural implementation. The processor's datapath is specified with a set of Prolog facts stating the architectural resources (registers, alu's, etc.) and its interconnections (buses). A special component connected to different points of the structural model collects the measurements needed to evaluate performances. Figure 2 and 3 show the high level specification for an instance of a instruction set and datapath for WAM. Note that WAM is a tagged architecture, that is, some registers and memory cells have two fields, tag and value.

![Figure 2](image1.png)

3. RTL operations Generation.
Instruction set high level specification is compiled into a RTL operation code where control structures, complex arithmetic expressions, auxiliary routines calls, and memory access have been removed. The target language consist of register transfers, simple arithmetic operations, and conditional and unconditional jumps. This language only makes reference to WAM's architectural elements, so it is independent of any specific architectural implementation.

Prolog allows to write compilers for formal languages in a succinct and readable manner using DCG notation. First, the VHDL program is parsed, by means of a DCG grammar, and its abstract syntax is created. For example, a DCG rule for parsing and analyzing a sequential assignment statement is written as follows:

\[
\text{statement(assign}(\text{Vari,Exp})) \rightarrow \\
\text{variable(Var), ['='], expression(Exp)}.
\]
% register(name, inputs, outputs, type).
register(constants, [], [bus1, bus2], const_gen).
register(hreg, [bus1], [bus1, bus2], value_reg1).
... 
register(enreg, [bus1], [bus1, bus2, x1], value_reg3).
register(arg2, [], [bus1, x2], value_reg2).
register(mar, [bus1, bus2, x3], [bus1, bus2], mar_type).
register(mdr, [bus1, bus2], [bus1, bus2], mdr_type).
... 
register(temp1, [bus1, bus2], [bus1, bus2], value_reg4).

% functional_unit(name, input1, input2, output, type).
functional_unit(alu1, x1, x2, x3, alu).
functional_unit(alu2, bus1, bus2, x4, alu).

% conex(name, type).
conex(bus1, value).
conex(bus2, tag_value).
conex(x1, value).

Figure 3 Example of datapath and its specification.

Then the RTL operations are generated using the abstract syntax and some complementary information. The code generator uses another DCG grammar for the target language, running in a backward way, that is, generating a sequence of RTL operations from the abstract syntax. This is possible because of the bidirectionality of Prolog predicates. Following there is a simplified example of a DCG clause for code generation for an if statement:

code_gen(if(Condition, Stm1, Stm2)) -->
    conditional_branch(Condition, label(n1), label(n2)),
    [label(n1)], code_gen(Stm1), [goto(label(n3))],
    [label(n2)], code_gen(Stm2),
    [label(n3)].

Figure 4 shows source VHDL code for WAM's machine instruction UNITVAR, and its corresponding generated RTL operation code. UNITVAR corresponds with a functor in the head of a Prolog clause, and when executed, if machine is in write status, a new cell pointed by the corresponding argument register is push into the heap; and if machine is in read status, the argument register becomes pointing to the appropriate position in order to perform unification of the arguments of the structure.
In this step, the following actions are performed:

### Datapath resources (buses and functional units) needed for its execution are assigned to each RTL operation. In general, there is many different forms to perform a RTL operation, each of them involving a different set of resources. At this point it is not possible to choose the best set of resources because, if we want an optimal solution, resource assignment must be done along with scheduling. So, in this step we just associate to each RTL operation a list with all its different alternatives for execution. Each element of this list is also a list which comprise the set of resources needed for execution of that alternative.

### Actual architecture registers are assigned to auxiliary variables and temporary storage.

### Number of control steps needed for its execution is assigned to each RTL operation.

### In order to establish data dependencies later while scheduling all source and destination registers involved must be determined.

Figure 5 shows the output from the preliminary resource assignment step for the sequential fragment label 1 from example in Figure 4.

```
UNITVAR:
if RW_mode = WRITE then
  MEMORY (HREG) := (ref,HREG);
  REGFILE (arg1) := (ref,HREG);
  HREG := HREG+1;
else
  REGFILE (arg1) := (ref,SREG);
  SREG := SREG+1;
end if;
fetch;
```

```
unitvar:
compare(RW_mode,write)
goto_if_else(equal,1,2)

label 1
kte(ref) => mdr.tag
HREG => mdr.value
HREG => mar
mdr => memory(mar)

else
  arg1 => regindex
  kte(ref) => regfile.tag
  HREG => regfile.value
  HREG+1 => HREG
  goto (fetch)
end if;
label 2:
kte(ref) => regfile.tag
SREG => regfile.value
SREG+1 => SREG
goto(fetch)
```

**Figure 4** RTL operation Generation.

**Figure 5** Preliminary resource assignment.
5. **Resource Assignment and Scheduling.**

Optimal planning involves both assigning a control step (scheduling) to each RTL operation and choosing one of its alternatives of execution, in such a manner that the total number of control steps is minimized. Scheduling must resolve conflicts emerging when two or more RTL operations need the same datapath resource, establishing the order in which resources will be used.

Constraint Logic Programming over finite domains extends Prolog with mathematical techniques to solve numerical constraints and consistency techniques to solve symbolic constraints. CLP allows a declarative expression of combinatorial problems and, in optimization problem like this one, constraints may be used to prune the search space in *a priori* way, thus improving run time efficiency [7].

Optimal resource assignment and scheduling problem is stated as an activity network [6] where nodes correspond to RTL operations and edges correspond to precedences imposed both because of data dependencies and resource sharing. A data dependence between RTL operations *i* and *j* imposes a directed fixed precedence constraint, that is, operation *i* must precede operation *j*. We model this relation by means of the following logic constraint:

\[
\text{precedence}(S_i, D_i, S_j, D_j) :- S_j \geq S_i + D_i.
\]

Where *S_k* is a logic domain variable which stands for the control step at which RTL operation *K* is going to be executed, and *D_k* stands for its duration.

Datapath resources are shared by the set of RTL operations with no data dependencies among them. Two RTL operations, say *i* and *j*, that need to use the same resource can not be executed at the same time, that is, operation *i* must precede operation *j*, or operation *j* must precede operation *i*. We model this relation by means of a disjunctive constraint:

\[
\text{disjunctive}(S_i, D_i, S_j, D_j) :- S_j \geq S_i + D_i.
\]

\[
\text{disjunctive}(S_i, D_i, S_j, D_j) :- S_i \geq S_j + D_j.
\]

Choosing one of the possible alternatives of execution for each RTL operation is modeled through the classical non-deterministic predicate *member/2*:

\[
\text{resource_assignment} \left( \text{Set_of_Resources}_k, \text{List_of_Alternatives}_k \right) :- \text{member} \left( \text{Set_of_Resources}_k, \text{List_of_Alternatives}_k \right).
\]

So, finding a resource assignment and scheduling solution is solved by the clause:

\[
\text{scheduling} \left( \text{List_of_Operations}, \text{List_of_S}_k \right) :- \text{build_domain_vars} \left( \text{List_of_S}_k \right), \text{data_dependencies} \left( \text{List_of_Operations}, \text{List_of_S}_k \right), \text{resource_assignment} \left( \text{List_of_Operations}, \text{List_of_Sets_of_Resources} \right), \text{disjunctives_precedences} \left( \text{List_of_Sets_of_Resources}, \text{List_of_S}_k \right), \text{labelling} \left( \text{List_of_S}_k \right).
\]
Predicate `data_dependencies/2` recursively makes use of `precedence/4` in order to establish the precedence constraints due to data dependencies. Predicate `resource_assignment/2` chooses a set of resources for each RTL operation. Predicate `disjunctive_precedences/2` recursively makes use of `disjunctive/4` in order to establish the disjunctive constraints due to resource sharing. Finally, `labelling/1` predicate assigns values, according with the constraints previously stated, for the domain variables of the list `List_of_Sk`. Each element of this list stands for the control step at which the corresponding RTL operation is executed.

Predicate `scheduling/2` yields a scheduling solution, and eventually via backtracking all possible solutions. But we are not looking for any solution, but the best one, that is, the one which minimizes the total number of control steps. Search space may be vast because of the presence of alternatives for execution of each RTL operation, and also because of choices introduced by predicate `disjunctive/4`. In order to find the best solution we make use of the optimization predicate, `minimize/2`, which finds the solution for its first argument (a predicate) which yields the minimum value for its second argument (a variable). So, `best_scheduling/2` is implemented as follows, where `Send` stands for the last RTL operation's control step:

```
best_scheduling(List_of_Operations, List_of_Sk) :-
    build_domain_vars(List_of_Sk, Send),
    data_dependencies(List_of_Operations, List_of_Sk),
    minimize(make_choices(List_of_Operations, List_of_Sk), Send).

make_choices(List_of_Operations, List_of_Sk) :-
    resource_assignment(List_of_Operations, List_of_Sets_of_Resources),
    disjunctive_precedences(List_of_Sets_of_Resources, List_of_Sk),
    labelling(List_of_Sk).
```

Built-in predicate `minimize/2` takes advantage of constraints capabilities for pruning search space in `a priori` way. The search strategy implemented within this predicate is as follows: we first search for a solution. Then, search is restarted from the beginning with an additional constraint: next solution must be better than the one already found. This process is repeated until no better solution can be found. This is a `branch and bound` method; as soon as a partial solution is found that is worse than the previous solution, the search is abandoned and a new solution is searched for.

Figure 6 shows the optimal resource assignment and scheduling obtained for the example of Figure 5. A final step is needed in order to code this information into the binary form which is stored in the ROM memory of the control unit.
OPERATION LISTS OF RESOURCES CONTROL DURATION

<table>
<thead>
<tr>
<th>label 1:</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A)</td>
<td>kte(ref) =&gt; mdr.tag</td>
<td>[bus2.tag]</td>
<td>1 2</td>
</tr>
<tr>
<td>B)</td>
<td>HREG =&gt; mdr.value</td>
<td>[bus1]</td>
<td>1 2</td>
</tr>
<tr>
<td>C)</td>
<td>HREG =&gt; mar</td>
<td>[bus1]</td>
<td>3 2</td>
</tr>
<tr>
<td>D)</td>
<td>mdr =&gt; memory(mar)</td>
<td>[]</td>
<td>5 3</td>
</tr>
<tr>
<td>E)</td>
<td>arg1 =&gt; regindex</td>
<td>[bus2.value]</td>
<td>1 2</td>
</tr>
<tr>
<td>F)</td>
<td>kte(ref) =&gt; regfile.tag</td>
<td>[bus2.tag]</td>
<td>3 2</td>
</tr>
<tr>
<td>G)</td>
<td>HREG =&gt; regfile.value</td>
<td>[bus2.value]</td>
<td>3 2</td>
</tr>
<tr>
<td>H)</td>
<td>HREG+1 =&gt; HREG</td>
<td>[]</td>
<td>5 2</td>
</tr>
<tr>
<td>I)</td>
<td>goto (fetch)</td>
<td>[]</td>
<td>7 2</td>
</tr>
</tbody>
</table>

Figure 6 Optimal resource assignment and scheduling.

6. Conclusions.

CAD system design involves two main tasks, translation from one level to another lower level, and the optimization of the results. In this paper we have demonstrated that Prolog's DCG and CLP are well suited for implementing in a declarative and efficient way the translation and optimization phases of LPCAD system. DCG's have been used for parsing and code generation, and CLP has been used for the optimal planning (resource assignment and scheduling) of the RTL operations resulting from compilation.

The automatic synthesis tool and design methodology described in this paper have been used to evaluate different design alternatives for a WAM based special purpose processor for Prolog execution. Architecture with different number of buses have been evaluated, showing that a two-bus architecture offers the best trade-off between cost and performance. Also, different processor's instruction sets and different kinds of registers have been tested. Future work will focus on lower levels of design to realize the first silicon prototype of the processor.

References