Using Logic Programming and Coroutining for VLSI Design

(Summary)

Ulrich Bieker, Andreas Neumann

University of Dortmund, Department of Computer Science

D-44221 Dortmund, Germany

e-mail: bieker@ls12.informatik.uni-dortmund.de

1. Abstract

We show how an extended Prolog can be exploited to implement different electronic CAD tools. Starting with a computer hardware description language (CHDL) several problems like digital circuit analysis, simulation and code generation for programmable microprocessors are discussed. For that purpose a part of the MIMOLA (machine independent microprogramming language) system MSS (MIMOLA hardware design system) is presented. Several advantages obtained by applying techniques of logic programming to solve problems in the area of integrated circuit design are shown. Especially maintenance, small source code, backtracking and the extension of standard Prolog by a coroutining mechanism to express Boolean constraints are pointed out.

2. Introduction

Due to the increasing complexity of digital circuits, the design process is supported by a lot of design tools covering a wide range of problems. A lot of these problems are of high complexity. Therefore electronic CAD systems, commonly written in imperative languages, consist of a very large amount of source code. Maintenance, portability and adaptability becomes a problem. We describe significant software engineering advantages by using Prolog for these problems.

MIMOLA [BMSJ91] is a computer language with Pascal-like constructs. It supports design, test, simulation and programming of digital computers and is integrated into the CAD system MSS [Marw84]. MIMOLA, influenced by other hardware description languages like VHDL [IEEE92], allows structural and behavioral descriptions of circuits. Originally the complete system has been written in Pascal but onwards from MIMOLA 4.0 we started to redesign several tools using Prolog.

Using the extended Prolog system ECLIPSE [ECRC92] new concepts to solve problems in the area of digital circuit design have been found. Coroutining, which allows the user to express a condition under which a call to a specified goal will be delayed, is a very useful mechanism to avoid unnecessary backtracking steps during simulation, test and code generation.

Several approaches to digital circuit design using logic programming have been presented [Gull85, Cloc87, Simo89, DSH90], most of them concentrating on the gate level or even lower levels of abstraction. Only a few contributions consider higher levels of abstraction in the context of logic programming [Rein91, LWG91].

In this paper we describe the use of Prolog for a very high level of abstraction. An elegant simulator, based on a hardware description language is presented. The simulator is able to simulate a processor together with a given program. We also present a concept to generate microcode for a given hardware structure.

Starting with a circuit given as a MIMOLA hardware description an adequate tree based Prolog circuit representation is generated from a frontend compiler. Afterwards, a circuit analyser creates a circuit info file that can be used as input for the codegenerator. Finally the generated microprogram can be simulated together with the circuit description.

In what follows we first introduce a small but illustrating processor used as an example for the whole paper. We continue with the simulator concept based on three levels of abstraction, followed by a section describing circuit analysis to prepare code generation.

3. SIMPLECPU: A simple processor

Figure 1 shows SIMPLECPU, a small programmable microprocessor consisting of 8 modules. The SIMPLECPU controller (shaded area) consists of a program counter, an instruction memory, an incrementer and a multiplexer. Furthermore a 16 x 4 register file, a 4 bit ALU, a second multiplexer and a clock are part of the structure. Register file and program counter are connected to the clock (not shown) and control signals are denoted by c followed by an index range. MIMOLA hardware descriptions contain register transfer modules, their behavior and their interconnections. For instance, the 4 bit ALU is specified in MIMOLA as shown below. CONBEGIN and CONEND denote a concurrent block, containing two case expressions as assignments to the outputs. In MIMOLA, the default data type is the bit vector. Its index range is denoted as (high-bit : low-bit), i.e. the ALU has two 4 bit data inputs *a* and *b*, a 4 bit output *result*, a 1 bit output *condition* and a 2 bit control input *ctr* selecting the ALU function.

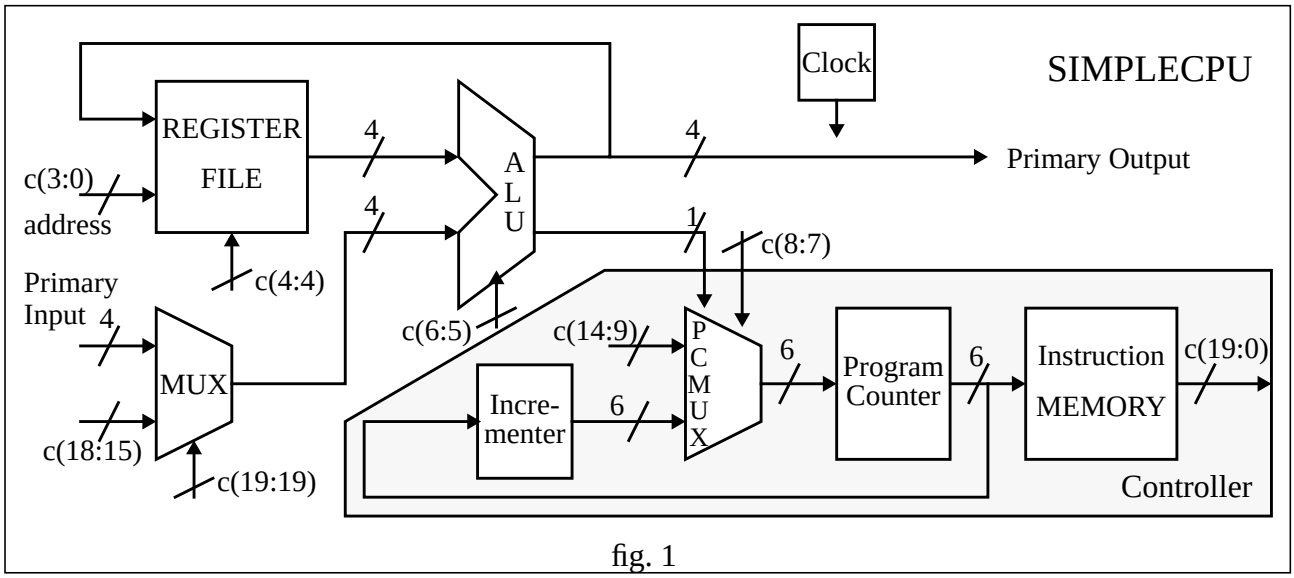

fig. 1

```
MODULE ALU
    (IN a, b : (3:0); IN ctr : (1:0);
     OUT result: (3:0); OUT condition:(0:0 ));
    BEHAVIOR IS
    CONBEGIN
        result <- CASE ctr OF
                0 : a ;
                1 : b ;
                2 : a+b ;
                3 : a-b ;
                END AFTER 1;
        condition <- CASE ctr OF
                0 : a = 0 ;
                1 : b = 0 ;
                2 : a+b = 0 ;
                3 : a-b = 0 ;
                END AFTER 0;
    CONEND;
```

Using MIMOLA as input language, we generate a tree based Prolog intermediate format. A circuit is represented as a list of module descriptions. Every module consists of a list of connections, a list of storing cells and a behavior tree. Such a tree is easily represented by a Prolog structure. The list of connections contains information about inputs and outputs of the module and interconnections to other modules. Every signal is represented by a logic variable and this variable also occurs in the behavior tree when the signal is referenced. If signals are instanciated elsewhere, this leads to an immediate signal propagation to all modules using this signal.

4. Simulation of a CHDL

The implemented simulator is based on three levels of abstraction: the built in operators, an interpreter for the behavior of a single component and an event driven simulator for a circuit together with a microcode. Especially for the implementation of the operators, we made extensive use of the coroutining concept of ECLIPSE. Due to the lack of space we do not consider the interpretation level as described in [Biek93] and the event driven simulation.

For the interpretation of a Hardware Description Language an implementation of its built-in operators is necessary, which range from logic primitives to complex arithmetic operators. These are represented as Prolog predicates, which mainly have to fulfill the following demands:

a) The operators must work bidirectionally, so that they can also be used for backward simulation of a circuit.

b) They should work deterministically, i.e. subsequent backtracking step must not produce the same solution. This is especially important for the backward simulation, as the mapping of an operator is not necessarily definitely reversible. Certain backtracking alternatives have to be pruned to avoid duplicate solutions.

c) The computation must be - at least at operator level - data driven, i.e. the application of an operator to unbound variables is propagated symbolically as a delayed goal, until the instantiation of the variables is absolutely unavoidable. By this, the number of backtracking steps is reduced.

The third point is achieved by using the coroutining mechanism of ECLIPSE, which allows the programmer to specify conditions, under which the execution of a goal shall be delayed, depending on the bindings of its parameters. Whenever a variable occuring in one of these is bound the goal will be woken, and the delay conditions are checked again.

Nevertheless, at the end of a simulation the set of all delayed constraints must be consistent, i.e. there should be a constraint solver[1] which finds contradictions and - if possible - solutions for variable bindings. Since such a constraint solver is rather

1. Actually we develop a Boolean constraint solver in Prolog on top of the ECLIPSE system.

complex, there should only be a few types of constraints. It would be sufficient to consider a minimal complete set of operators, but for efficiency reasons we used a set containing AND, OR, XOR and NOT. The Prolog code for those operators is now divided into delay clauses and program clauses, e.g. the logical AND is implemented as shown below:

```
delay and (X,Y,Z) if var(X), var(Y), var(Z), X\==Y.
delay and (X,Y,Z) if var(X), var(Y), Z==0, X\==Y.

and (X,Y,Z) :-
        nonvar(Y), !, and1(Y,X,Z).
and (X,Y,Z) :-
       nonvar(X), !, and1(X,Y,Z).
and (X,X,X).

and1 (0,_,0).
and1 (1,X,X).
```

The delay clauses cover the case, when the two inputs parameters X and Y are distinct unbound variables, and the output parameter is either unbound or zero. In these cases it is impossible to draw any conclusion, so the call to the predicate is delayed. The program clauses use the commutativity of the logical AND: the first two of them deal with the case, when one of the inputs is bound, and call *and1* with this one in the first place. For the third clause there are - due to the delay clauses - only two possibilities left: either the output is 1, which forces the inputs to take the same value, or the two inputs are identical variables, to which the output will be bound, too. The auxiliary predicate *and1* expect its first input to be instantiated. If it is bound to a 0, the result must be 0 either, if it is 1, the output is identical to the second input.

The more complex operators are now based on these

```
halfadd (In1,In2,Sum,Cout) :-
        and(In1,In2,Cout),
        xor(In1,In2,Sum).

fulladd (In1,In2,Cin,Sum,Cout) :-
        halfadd(In1,In2,Sum1,Carry1),
        halfadd(Cin,Sum1,Sum,Carry2),
        or(Carry1,Carry2,CarryOut).
```

four logical primitives, e.g. a full adder is defined as shown above. Of course the set of operators is not restricted to single bit operations, but for each of them there is also a version for bitstrings, which are represented as lists. Upon these are arithmetic operators like addition and multiplication and string manipulation operators like shifting and concatenating.

5. Circuit Analysis

When simulating a circuit it is necessary to give priorities to different modules concerning the order in which to simulate two events for the same time. The reason for this are causal dependencies between components which are connected without delay. This priority can be compared to the Δ-delay of VHDL. The intention is that an event may be simulated only when all events its inputs depend on have been considered before, i.e. the priority of a module is the maximum of the priorities of all its predecessors incremented by one. Assume that we have already computed a priority list of triples *(Mod, Prio,Preds)*, where *Pred* is a list of pairs *(Mod', Prio')*, so that every occurence of a module in the whole structure have its *Prio* component bound to the same variable. Now, for each element of the priority list, we only have to compute the maximum priorities in the predecessor list and bind the priority to this value incremented by one.

```
delay max(A,B,M) if var(A).
delay max(A,B,M) if var(B).

max(A,B,A) :- A > B,!.
max(A,B,B).

maxPriority([],Max,Max).
maxPriority([(_,Prio)|Rest],Max0,Max) :-
    max(Prio,Max0,Max1),
    maxPriority(Rest,Max1,Max).

setPriorities([]).
setPriorities([(Mod,Prio,Preds) | Rest]) :-
    maxPriority(Preds,0,MaxPrio),
    plus(MaxPrio,1,Prio),
    setPriorities(Rest).
```

In standard Prolog this would lead to difficulties because we could not compare unbound variables. This is easily resolved by delaying the *max* predicate. If there are no critical races in the circuit, i.e. there are no cyclic dependencies, there must be at least one module whose predecessor list is empty, so it will get priority 1. This will wake up at least one other *max* goal, and so on, so that all priorities will be computed correctly. If there is a cycle, then a conflict occurs and an error must be raised. Such a conflict can easily be detected by checking for delayed goals by a system call. Note that the plus/3 predicate must also be delayed, which is done automatically by ECLIPSE.

To prepare microcode generation, several tasks are done by the circuit analyser. The main task is to generate a lot of facts describing special characteristics of a given circuit to reduce complexity of code generation. Table 1 gives an overview of some generated facts but due to the lack of space not all generated facts can be considered.

The first fact is transparent/3, denoting an identity mapping from one input to at least one output, so that the module becomes ‘transparent’. That means that with a special control code, the considered module is able to pass one input to one output. The transparent/3 example of table 1 shows a possibility to switch input *a* of the SIMPLECPU *alu* to the output *result*, i.e. the signal list [D,C,B,A] is switched. This is done by unifying input *b* with the neutral element [0,0,0,0],

to perform an identity mapping for the selected operator. The binary control code c(6:5) = [1,0] selects the add operator.

Using the interpreter and the operators of the simulator, the transparent/3 facts are easily generated. Basic idea is to unify one input with one output and to perform an interpretation step for this module. The interpretation step has to lead to instantiations of some inputs for the following reasons:

a) Choosing of a control code to select an operation that is able to perform an identity mapping (e.g. c(6:5) = [1,0] to select ALU addition).
b) If necessary, choosing of a neutral element for the selected operation (some operations do not need a neutral element, e.g. the ALU operation selected by the control code c(6:5) = [0,0] to switch input *a* to the output *result*).

A successive selection of all operations performed by a module is done by backtracking. Afterwards, the selected operation has to be executed symbolically, holding the input port to be switched as list of variables. Execution of the selected operation (Op) is done by the clause findTransparent/4. The lists libr-

```
findTransparent(Module,Op,InPorts,OutPorts):-
    member(SwitchPort, InPorts),
    member(SwitchPort, OutPorts),
    Operation =.. [Op, InPorts, OutPorts],
    call(Operation),
    checklist(var, SwitchPort),
    assert(transparent(Modul,InPorts,OutPorts)),
    fail.
findTransparent(_,_,_,_).
```

ary predicate checklist/2 succeeds if var/1 succeeds for every element of SwitchPort, ensuring that the selected input is switched to the selected output for all possible values of SwitchPort. Finally, we assert the generated fact.

The fact path/3 describes a path from a source module to a destination module, possibly through certain other modules which are able to perform an identity mapping. A fact path/3 is a triple with parameters source, destination and *Path*. *Path* is a list of triples (module name, list of inputs, list of outputs). The first element of the list is the source module whereas the last element is the destination module. All modules between source and destination are able to switch an input to an output by the use of transparent/3. A path/3 fact contains all control codes, i.e signals which have to be 0 or 1 to switch the *Path*. The example given in table 1 shows a *Path* from the instruction memory *im* through the multiplexer *mux* and the *alu* to the register file *reg*. Therefore binary control codes c(19) = [0] for the multiplexer and c(6:5) = [0,1] to switch a via through the *alu* are selected. [D,C,B,A] is the list of values connected by this path.

A simplified version of the predicate generating path/3 facts is findPath/3. The first clause terminates the search of a path if *Destin* is a direct successor of Source. In the second clause we try to find a path through a module *Next*, which has to be a successor of the current *Source* and has to be switched into a transparent mode. Afterwards, a recursive search with *Next* as source is started. A lot of implementation

```
findPath(Source, Destin, [Source, Destin]):-
    successor(Source, Destin).

findPath(Source,Destin, [Source | RestPath]):-
    successor(Source, Next),
    transparent(Next, Inputs, Outputs),
    findPath(Next, Destin, RestPath).
```

details are omitted, e.g. the check to prevent entering a cycle and the complete circuit representation.

A frequently done subtask of microcode generation is to increment the program counter. Therefore we generate a symbolic increment instruction where the address is unbound. The real address will be instantiated at the end of code generation. For that reason we generate a delayed goal, so that the code generator is

Table 1: Some selected facts, generated by circuit analysis

fact/arity	arguments	example
transparent/3	module name list of inputs list of outputs	transparent(alu, [[D,C,B,A], [0,0,0,0], [1,0]], [[D,C,B,A], [Condition]]).
path/3	source destination Path	path(im,reg,[(im,[[_,_,_,_,_,_]],[[0, D,C,B,A,_,_,_,_,_,_,_,_,_,0,1, _,_,_,_,_]]), (mux, [[_,_,_,_], [D,C,B,A], [0]], [[D,C,B,A]]), (alu, [[_,_,_,_], [D,C,B,A], [0,1]] , [[D,C,B,A], [_]), (reg, [[_,_,_,_], [D,C,B,A], [_], [_]] , [[_,_,_,_]])]).
incrementPC/2	delayed goal Path	incrementPC(incr([F,E,D,C,B,A], [L,K,J,I,H,G]), [(inc, [[F,E,D,C,B,A]], [[L,K,J,I,H,G]]), (pcmux, [[_], [0,0], [L,K,J,I,H,G], [_,_,_,_,_,_]], [[L,K,J,I,H,G]]), (pcreg, [[L,K,J,I,H,G], [_]], [[F,E,D,C,B,A]])]).

able to bind these addresses to real values with respect to certain constraints. As a consequence of that, an increment instruction incrementPC/2 is a pair, containing a delayed goal which performs the increment operation and a *Path* from the output of the program counter to the input of the program counter. *Path* is a list of triples as described above. The given example of table 1 shows the unique solution to increment the program counter *pcreg* for the example processor. Therefore the binary control code c(8:7) = [0,0] is selected for the multiplexer *pcmux*. [F,E,D,C,B,A] is the current state of the program counter whereas [L,K,J,I,H,G] will be the next state. The delayed goal incr([F,E,D,C,B,A],[L,K,J,I,H,G]) denotes the operation to be executed at the end of code generation.

We conclude this section by enumerating some additional facts not considered here:

a) jump/1: denotes an unconditional jump, i.e. a possibility to move a constant value into the program counter without consideration of a condition.
b) conditionalJump/2: denotes a conditional jump version, i.e. a conditional path to the program counter.

These facts are mainly generated by the use of path/3 and transparent/3. Using failure driven loops (see e.g. findTransparent/4), all possible solutions of the described facts are generated and asserted.

At the end of code generation the microprogram has has to be bound to real addresses of the instruction memory. This is done by unifying the symbolic address of the first instruction with the start address e.g. 0. Now all delayed goals like incr/2 are woken and this leads to a successive binding of concerned addresses. Such microinstructions or even complete microprograms can be simulated by the simulator described above.

6. Conclusion

We have presented how logic programming and coroutining are exploited for some tools of the MIMOLA hardware design system. A simulator for structural hardware models, described in a hardware description language, has been presented. The simulator e.g. consists of 2700 lines of code whereas the original Pascal simulator has about four times more lines of code. Most of the new simulator can be used bidirectionally and symbolically which is very important for code and test generation. Using coroutining to express certain constraints, a lot of backtrakking steps can be avoided. Backward simulation in general is non-deterministic and therefore backtrakking and bidirectionality of Prolog is advantageous. Moreover, the original simulator is very difficult to maintain. Time to develop VLSI tools using logic programming is much shorter than for imperative languages. On the other hand, software written in standard Prolog is slower, but with the new concept of constraint logic programming this disadvantage becomes smaller, because this technique leads to a significant reduction of unnecessary backtracking steps.

This work was supported by the DFG, the German research foundation.

7. References

[BMSJ91] R. Beckmann, P. Marwedel, W. Schenk, and R. Jöhnk. The MIMOLA Language Reference Manual - Version 4.0. Research Report 401, Computer Science Dpt., University of Dortmund, February 1991.

[Biek93] U. Bieker. On the Formal Semantics of a CHDL - A Case Study. GI/ITG-Workshop: Formale Methoden zum Entwurf korrekter Systeme, Bad Herrenalb, March 1993.

[Clock87] W. F. Clocksin. Logic Programming and Digital Circuit Analysis. Journal of Logic Programming, pp. 59 - 82, March 1987.

[DSH90] M. Dincbas, H. Simonis, P. Van Hentenryck. Solving Large Combinational Problems in Logic Programming. J. Logic Programming, 1990.

[ECRC92] ECLIPSE 3.3 User Manual. ECRC Common Logic Progamming System. ECRC GmbH, Arabellastr. 17, Munich, Germany, August 1992.

[Gull85] E. Gullichsen. Heuristic circuit simulation using PROLOG. North-Holland, Integration, the VLSI-Journal, No. 3, pp. 283 - 318, 1985.

[IEEE92] Design Automation Standards Subcommittee of the IEEE. Draft standard VHDL language reference manual. IEEE Standards Dpt., 1992.

[LWG91] Y. Lichtenstein, B. Welham, A. Gupta. Time Representation in Prolog Circuit Modelling. 3rd UK Annual Conference on Logic Programming, Edingburgh 1991.

[Marw84] P. Marwedel. The MIMOLA Design System: Tools for the Design of Digital Processors, Proc. 21st Design Automation Conference, pp. 587 - 593, 1984.

[Rein91] P. B. Reintjes. A Setof Tools for VHDL Design. Logic Programming, Proc. of the eigth Int. Conference, pp549 - 562, 1991.

[Simo89] H. Simonis. Test Generation using the Constraint Logic Programming Language CHIP. In Proceedings of the 6th International Conference on Logic Programming, Lisboa, Portugal, pp. 101 - 112, June 1989.