A LOGIC PROGRAMMING APPROACH FOR DSP ARCHITECTURES DESIGN

Magdy A. Bayoumi and Khaled M. Elleithy

The Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504, U.S.A.

Abstract

This paper introduces a logic programming approach for specifying, simulating, and testing Digital Signal Processing (DSP) systems. Prolog is used as a Hardware Description Language and a host one, too. Backtracking and pattern matching of Prolog are employed for simulation and testing, respectively. Prolog provides homogeneity to the developed system as it supports hierarchical development and mixing of description at various hierarchical levels. The developed system belongs to Algorithmic Specific CAD family. It can be employed for many DSP algorithms and applications development.

1. Introduction

The advances in VLSI fabrication and packaging technologies have led to having very complex systems of 50,000 to 600,000 transistors on a single chip. Designing, testing, and verifying these chips using some of the common ad-hoc design techniques can take a lifetime. As a result, cost-effective VLSI design methodologies have become vital to produce correct chips which meet the corresponding specifications without exhaustive iterations and redesign. The difficulty of chip designs that it involves several levels of design; starting from chip level specifications to the geometrical layout. Automating the whole design spectrum has become one of the main goals. The lowest level of this spectrum is understood sufficiently well to enjoy a fair amount of automatic, while, developing design methods for the high levels is still in its infancy stage. Developing these design methods can be very expensive, to reduce such cost, the application specific design methods and CAD tools\[1,2\].

In this paper algorithmic specific IC's design methods and CAD tools are developed for Digital Signal Processing algorithms and architectures. The available DSP CAD tools can be classified into three categories. Firstly, is the silicon compilers in which the tools can transform high level description into layout\[3\]. Most of these compilers have constraints on the output system. The second approach depends on a set of special purpose cells used in building block cells. The third approach is to develop development systems\[2\], the main problem of these development systems is to have heterogeneous environment and modeling languages at all stages: specification, simulation, and verification.

Using PROLOG as a VLSI modeling language at all levels is the focus of this paper. Hardware Description Languages such ISP!\[4\] are capable of modeling circuits at the architectural and instruction levels. But, they differ from their simulation environments with the consequence that the approach is non-uniform. Communication and scheduling of hardware models are centralized and, consequently, inappropriate for multiprocessor designs. Several efforts for using procedural languages have been reported\[5,6\]. The problems with many of the efforts in VLSI specification languages are: attention to only a limited stages of the design spectrum, avoidance of the performance specification problem, lack of constructs to specify parallelism, and the lack of supporting the formal design approach.

Prolog\[7\] provides two important characteristics that other conventional languages lack: pattern matching and backtracking. Pattern matching is an appropriate tool for testing and backtracking is suitable for simulation. Prolog has powerful facilities for simulation. Using the same code without any modifications we can perform different types of simulation: forward, backward or bidirectional. Simulation can be performed on different levels and it can be mixed. Circuits specified in Prolog can be specified at different levels, mixed levels and any degree of details without too much complexity.

2. Circuit Specification Using Prolog

Prolog is a logic programming language that has been targeted as the language of the Japanese fifth generation computer research effort. Prolog statements are a subset of first order predicate calculus, referred to as horn clauses. There are basically two types of statements: facts and rules. Facts are simple true statements. Rules are implications. Here are examples of facts:

\begin{itemize}
  \item \texttt{man(john).}
  \item \texttt{son(jeff,john).}
\end{itemize}

An example of a fact:

\begin{itemize}
  \item \texttt{father(X,Y):-man(X),son(Y,X).}
\end{itemize}

We have two facts and one rule. Constants start with lower case letters and variables start with upper case letters. The facts we have state that \texttt{john} is a man and \texttt{jeff} is the son of \texttt{john}. We can add any other facts about all men that we know or persons we know their sons. The rule we have states that: \texttt{X is the father of Y if X is a man and Y is the son of X}. The set of all facts and rules is called the database, which is parallel to the notation of a program in a conventional language.
One other form of statements which triggers the program to be executed in the question. For example to answer the question father (john, jill), the Prolog system will try to match this question with the facts and rules in the database. The question which is required to be satisfied is called the goal. Each goal can be divided into a number of subgoals. Each subgoal is satisfied in the same order determined by the used rule. Satisfying subgoals will bound the unknown variables to specific values, this is called instantiation. If a subgoal cannot be satisfied, backtracking is used to resatisfy the previous subgoals. To eliminate unnecessary backtracking the cut operator can be used, which is represented by ( ! ). The use of the cut operator will make programs faster and reduce the required memory.

Prolog does lend itself for circuit representation. Since prolog is a logic programming language, it stresses problem definition instead of devising algorithms for problem solving (like traditional languages). The following is a Prolog segment which represent an AND gate, lit+.

First two arguments are the inputs, and the third argument is the output:

```prolog
and(0, 0, 0).
and(0, 1, 0).
and(1, 0, 0).
and(1, 1, 1).
```

On examination of the above example, it would appear that it is a very simple matter to describe any circuit module directly from a truth table. Of course once primitive gates are defined using Prolog facts, it is possible to define more complicated circuits using these primitive gates. As an example the Prolog representation of the half adder is shown in Figure 1.

```prolog
half_adder (Input_1, Input_2, Sum, Carry):-
or (Input_1, Input_2, Or_output),
and (Input_1, Input_2, Carry),
not (Carry, Not_output),
and (Or_output, Not_output, Sum).
```

The hierarchical description that Prolog allows can be extended to any level. The subgoals in the implications can be other implications, as well as facts. We will show this by an example of a full adder. First we introduce the full adder representation using primitive gates (Figure 2).

```prolog
full_adder (Input_1, Input_2, Carry_in, Sum, Carry_out):-
or (Input_1, Input_2, Or_1_output),
and (Input_1, Input_2, And_1_output),
not (Carry_in, Not_1_output),
and (Or_1_output, And_1_output, And_2_output),
not (Carry_out, Not_2_output),
and (not_1_output, not_2_output, Sum).
```

As we see from the second representation of the full adder that circuit representation can be built up in a structured, hierarchical manner from simpler components. The results are easy to read, and concise.

The second method that can be used to represent the full adder is to use the half adder as a primitive building block as shown in Figure 3.

```prolog
full_adder (Input_1, Input_2, Carry_in, Sum, Carry_out):-
half_adder (Input_1, Input_2, Sum_1, Carry_1),
half_adder (Carry_in, Sum_1, Sum, Carry_2),
or (Carry_1, Carry_2, Carry_out).
```

The hierarchical description that Prolog allows can be extended to any level. The subgoals in the implications can be other implications, as well as facts. We will show this by an example of a full adder. First we introduce the full adder representation using primitive gates (Figure 2).

```prolog
full_adder (Input_1, Input_2, Carry_in, Sum, Carry_out):-
or (Input_1, Input_2, Or_1_output),
and (Input_1, Input_2, And_1_output),
not (Carry_in, Not_1_output),
and (Or_1_output, And_1_output, And_2_output),
not (Carry_out, Not_2_output),
and (not_1_output, not_2_output, Sum).
```

As we see from the second representation of the full adder that circuit representation can be built up in a structured, hierarchical manner from simpler components. The results are easy to read, and concise.
Because of the manner in which Prolog binds variables the representation of circuits can be reduced. Consider the representation of a simple AND gate, since a 0 on any input determines the output, it is possible to represent the AND gate in the following manner:

\[
\text{and}(0, X, 0).
\]
\[
\text{and}(X, 0, 0).
\]
\[
\text{and}(1, 1, 1).
\]

In the previous representation \( X \) can take a value 0 or 1. This technique is used to minimize backtracking to the lowest possible degree which has a great implication on the final performance.

To further limit backtracking, the predicates should succeed if no variables are bound. Indeed this success should occur before any arbitrary binding is performed by the predicate. This will prevent unnecessary backtracking. To accomplish this, we can add the following rule to the \( \text{AND} \) gate representation before the other facts:

\[
\text{and}(X, Y, Z) :- \text{free}(X), \text{free}(Y), \text{free}(Z).
\]

The \text{free} predicate is a library predicate that returns true if the argument is not bounded. If \( X \), \( Y \), and \( Z \) are not bounded in the previous rule, then \( Z \) will not be bounded. Because the previous rule is the first rule this prevents any other rules to be examined.

\section{Circuit Simulation}

Simulation can be performed in different manners in Prolog: forward, backward, or a combination of forward and backward simulation (bidirectional). Forward simulation can be defined as providing the input(s) to a circuit, with the simulation providing the output(s) that the inputs would produce. Backward simulation can be defined as providing the output(s), with the simulation providing one (or all) possible set(s) of input(s) that would produce the given output(s).

Circuits specified (defined) in Prolog can be simulated directly. In addition forward, backward, and bidirectional simulations can be performed in the same amount of complexity in Prolog with no additional specifications (unlike other traditional languages). If we use the previous example of the \text{half-adder}, forward simulation can be achieved by providing the following question to the simulator:

\[
\text{:-half_adder}(1,0,\text{Sum},\text{Carry}).
\]

The answer is:

\[
\text{Sum} = 1, \\
\text{Carry} = 0, \\
\text{yes}.
\]

Using the same code of the \text{half_adder} we can have backward simulation, here is an example:

\[
\text{:-half_adder}(X,Y,1,0).
\]
\[
X = 1, \\
Y = 0, \\
\text{yes}.
\]

The semicolon in the previous example is used to check for more answers. Since in this example we provide the outputs which are: \( \text{Sum} = 1 \) and \( \text{Carry} = 0 \), we get two sets of answers, which are exactly all the possible set of answers. Here is another example of bidirectional simulation, where you provide the system with a subset of the inputs and a subset of the outputs, the answers for this will be the possible unspecified set(s) of inputs and outputs that satisfy the given input, here is an example:

\[
\text{:-half_adder}(X,0,1,\text{Carry}).
\]
\[
X = 1, \\
\text{Carry} = 0, \\
\text{yes}.
\]

no more answers.

In the example of bidirectional simulation we only get one set for the answer, which is the only possible set.

The final example we use here is to show how to use Prolog in testing and verification. To test that our design is correct, we have to examine all the sets of inputs and outputs. In case of the \text{half_adder} we have four possibilities (\text{truth_table}); here is the testing example:

\[
\text{:-half_adder}(0,0,0,0). \\
\text{yes}. \\
\text{:-half_adder}(0,1,1,0). \\
\text{yes}. \\
\text{:-half_adder}(1,0,1,0). \\
\text{yes}. \\
\text{:-half_adder}(1,1,0,1). \\
\text{yes}.
\]

\section{ROM Implementation}

A read only memory has a fixed set(s) of predefined functions. The output is the value stored at the given location. \text{ROM} is a very important module in DSP applications. In the DSP we build in the last section the \text{ROM} is used to store the constants.

We will examine three different ways to implement \text{ROM}.

In the first method the simulator will compute the function which is stored in the \text{ROM}. The user sends in a list of inputs and a variable to hold the answer, for example \text{rom}([1,1],C). The calculating method is simple, the power of the bit's position is computed and added with similar values.

\[
\text{rom}([1,1]). \\
\text{rom}([0,0]). \\
\text{rom}(X,\text{Val}) :- \text{length}(X,M), N \text{ is } M-1, \\
\text{cal}(X,N,\text{Val}).
\]

The following procedure calculates the value of the bit by multiplying with 2th power and then adding with the previous value:

\[
X = 0, \\
Y = 1, \\
\text{yes}; \\
\text{no more answers}.
\]
Multiplier Implementation

The multiplier is implemented in two ways. In the first method we use the traditional add-shift method. The multiplicand and the multiplier are sent as two lists to the multiplication routine. The multiplier is reversed since the implementation scans the multiplier from left to right. The multiplication is done by examining the multiplier bits from right to left. If the scanned bit is one then the multiplicand is added to the partial sum which is initialized by 0. If the scanned bit is zero nothing is added. After the addition step a shift step is required.

In the second approach the routine performs an operation of the least significant bit of the multiplier with the multiplicand and this is the answer for the first place. Then the routine makes a list of zeros of a size equal to the multiplier length. After this a different method of addition is used. The carry from one full adder is sent as input to the next adder. The complete simulator is shown in Table 1.

5. FIR Filter Implementation

As a case study, consider the implementation of a Finite Impulse Response (FIR) filter. Figure 4 shows a systolic FIR structure in which the building block module is an inner product cell. The input is sent to a series of multipliers as the multiplicands. The other input for each multiplier is different. The input is fed to the first multiplier without a delay, to the second multiplier with a delay D, to the third multiplier with 2*D,..., for the n-th multiplier with (n-1)*D. There are n adders such that for adder i: one input is from multiplier i and the other input is from adder i + 1. For adder n one of the inputs is 0.

The simulator has as an input the number of circuits used and the constants used by the DSP. The following routine is the input routine:

dsp (ANS):-write ('Enter No of circuits'), read (No ),nI,
write ('Enter Input'),read (In ),nl,
write ('Enter List of multipliers'),read (Mpl),d,
write ('EnterDelay -ms -'),read (Time ),nl,
circuit (No ,In ,Mpl, Time ).

The following is the circuit simulation routine which determines the number of circuits and calling the multiplier, adder and delay routines:

circuit (0,-,0,0,0,1,49).
circuit (N,X,(H I T ,New,Time )):-
M is N-1,
circuit (M ,X ,T ,Out ,Time ),
multiplier (X ,[H],Res ),
adder (Res ,Out ,New,Sh ),
delay (Time,X).

The following is the circuit simulation routine which determines the number of circuits and calling the multiplier, adder and delay routines:

circuit (0,-,0,0,1,49).
circuit (N,X,[H|T],New,Time ):-
M is N-1,
circuit (M ,X ,T ,Out ,Time ),
multiplier (X ,[H],Res ),
adder (Res ,Out ,New,Sh ),
delay (Time,X).

Finally we need to represent the delay element. The following routine delays the input the requested amount of time:

delay (0,X).
delay :-M is N-1,
delay (M,X).
7. Concluding Remarks

The Prolog database mechanism can record both functional and physical characteristics of primitive gate types and user-defined modules. A simulator developed in Prolog enjoys a high degree of flexibility; inputs and outputs of a Prolog predicate need not be specified a priori. Further, heuristics can be written in Prolog to guide the simulator; thus reducing the required backtracking. Forward, backward, and bidirectional simulations are specified as easily. The ability to efficiently perform backward simulations is useful for circuit's testing. A hierarchical abstraction of circuit modules may be achieved, which makes the simulation of complex circuits tractable. For multiprocessors and concurrent systems, concurrent Prolog will be a powerful tool.

Acknowledgment

The authors wish to acknowledge Tulsiram Ravuri and John Varderh for their assistance in this project.

References


![Figure 4: A Systolic FIR Filter Structure](image_url)

Table 1: A Prolog Multiplier Simulator

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>Multiplier</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mpcnd.Mpl</td>
<td></td>
<td>Ans</td>
</tr>
<tr>
<td></td>
<td>Mpl</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mpcnd.Mpl</td>
<td>Ans</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ans</td>
</tr>
</tbody>
</table>

/* this part reverses a list of elements and creates a new list */
multiply(Mpcnd.Mpl) -> reverse(Mpcnd.Mpl),
multiply(Mpcnd.Mpl,Ans),
write('Ans is '),write(Ans).

/* this part adds two lists of elements, by first reversing them and */
/* reversing the answer */
add([Xn,Yn],Ans,Sh) -> reverse([Xn,Yn],Xn),add(Xn,Xn,Ans,Sh),reverse(Ans,Ans).
add([Xn,Yn],Ans,Sh) -> reverse([Xn,Yn],Xn),add(Xn,Xn,Ans,Sh),reverse(Ans,Ans).
add([Xn,Yn],Ans,Sh) -> reverse([Xn,Yn],Xn),add(Xn,Xn,Ans,Sh),reverse(Ans,Ans).