Translation from Transactions of IPSJ

Reasoning for a Logic Circuit Synthesis Expert System

KEISUKE BEKKI*, TAKAYOSHI YOKOTA*, TOHRU NAGAI*,
NORIKO NAKATSUKA* and NOBUHIRO HAMADA*

Most first-generation expert systems are rule-based systems with separate inference engines. To build this type of expert system, experts' knowledge must be described as a large unstructured collection of rules. But this is especially difficult for complicated design problems such as logic circuit design. Therefore a reasoning mechanism must be established that is suited to building expert systems for design problems.

This paper shows that the logic circuit design process can be decomposed into two phases. The first includes routine design, which is largely top-down, while the second is bottom-up. A new reasoning mechanism is proposed that can control both top-down and bottom-up design. This reasoning mechanism has been implemented in Prolog according to an object oriented programming paradigm, and an expert system called ProLogic built for logic circuit design. Logic circuits synthesized by ProLogic are evaluated and found to be similar to those designed by human experts, confirming the usefulness of the proposed reasoning mechanism.

1. Introduction

In order to shorten the design period for VLSI chips, their design automation has been researched extensively in recent years. High-level synthesis such as architectural and register-transfer-level synthesis, however, is not automated and is done manually. This is because high-level synthesis is a complicated procedure with the following characteristics:

1. A design object has a hierarchical structure;
2. Top-down design, in which lower layers are designed by using the design results of upper layers, and bottom-up design are flexibly combined to synthesize logic circuits;
3. When a solution that satisfies the specifications cannot be found, the design is reattempted, using another design strategy;
4. Design methods vary along with functional modules;
5. Design methods are modified according to the progress of design object fabrication technology.

These characteristics are common to almost all design problems. Therefore, building expert systems for design problems is difficult, and few practical systems have been reported.

Most expert systems are rule-based systems with separate inference engines. Experts' knowledge must be represented as a large unstructured collection of rules. Using these rules, the expert systems derive solutions. Usually, rules are expressed by a simple *if-then* type of structure. However, since the structure's knowledge representation ability is relatively low, it is very difficult to create an exact representation of the knowledge for solving complicated problems. Therefore, in building expert systems for such complicated problems, there is a gap between the complexity of the problem to be solved and the knowledge representation ability offered by an expert system. This gap must be filled by knowledge engineers, and poses the most difficult problem in building expert systems.

Our proposal for solving this problem is the concept of generic tasks [7, 8]. Most problem-solving methodologies can be classified into six types of tasks, called generic tasks, and expert systems for a specified problem domain can be built easily by combining the most appropriate tasks. Analysis of design problems shows that they can be classified into three classes:

1. Design problems in which neither design strategies nor design elements are given;
2. Design problems in which design elements but not design strategies are given; and
3. Design problems in which design strategies but not design elements are given.

For class 3 design problems where top-down design is possible, a generic task, called a routine design, was formulated in which the activity is *plan selection and refinement.*

We have been researching the methodology of building expert systems for high-level synthesis of VLSIs [4–6] belonging to class 3. However, since the search space for selecting design elements is huge, high-level synthesis cannot be solved without taking a bottom-up approach as well as a top-down approach. Thus, we cannot obtain a satisfactory solution for

This is a translation of the paper that appeared originally in Japanese in Journal of IPSJ, Vol. 31, No. 6 (1990), pp. 753-762.
*Hitachi Research Laboratory, Hitachi Ltd. 4026, Kuji, Hitachi, Ibaraki 319-12, Japan.

VLSI design by using a routine design provided by an expert system. Therefore, in order to build expert systems for design problems more easily, we must extend the concept and functions of the previously formulated generic tasks.

In this paper, we discuss a high-level synthesis that generates logic gate circuits from register-transfer-level specifications. We show that high-level synthesis can be achieved by taking both top-down and bottom-up approaches. Then, we discuss a reasoning mechanism to control the use of top-down and bottom-up approaches properly. Moreover, we show that this reasoning mechanism can be implemented easily by introducing the object-oriented programming paradigm, and that expert systems for high-level synthesis can be easily implemented by using the proposed reasoning mechanism.

2. High-Level Synthesis for VLSIs

Logic circuits such as microprocessors and digital signal processors consist of various kinds of functional modules. Figure 1 shows an example block diagram of a microprocessor. Its functional modules include an Arithmetic Logic Unit (ALU), a shifter, and a micro program sequencer. Functional modules can be grouped into two types: modules that were previously designed and can be used without modifying any parts, and modules that were previously designed and can be used after modifying some parts. From this viewpoint, we can classify the high-level synthesis problems into the following two classes:

Class A: Problems that can be solved by using pre-designed functional modules without any modification.

Class B: Problems that can be solved by modifying pre-designed functional modules.

Design problems belonging to Class A can be solved by using a routine design that takes only a top-down approach. Figure 2 shows the structure of a stack belonging to a Class B problem. The stack consists of a counter and a Random Access Memory (RAM). we assume that the counter and the RAM, which have been designed previously, can be utilized without any modification. The procedure of designing the stack then consists of two steps, constraint propagation and design element selection. The first step propagates many kinds of design constraint, imposed on functional modules of higher layers, to functional modules of lower layers. In the second step, the functional modules that satisfy all the design constraints are chosen from the functional module library that includes the previously designed functional modules. For instance, suppose that the specifications of the stack are given. At the constraint propagation step, the design constraints such as bit width, delay time, size, power consumption, clock scheme, and fabrication process are propagated to the modules of the lower layer, namely the RAM and the counter. At the design element selection step, the most suitable RAM and counter, which satisfy all the constraints propagated from the higher layer, are selected from the functional module library.

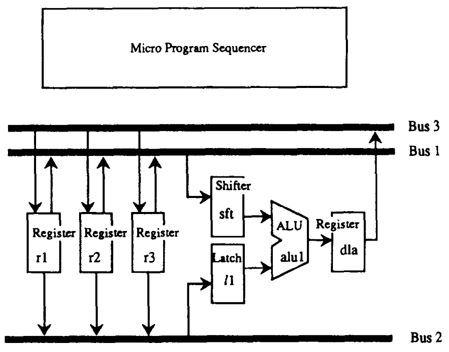

Fig. 1 Block diagram of a microprocessor.

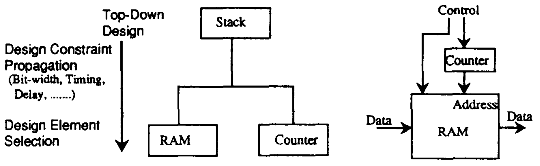

Fig. 2 Component structure of stack.

Design problems belonging to Class B can be solved by using both top-down and bottom-up approaches appropriately. Figure 3 shows the hierarchical structure of a microprocessor. Each box represents a functional module. Boxes marked '*' represent functional modules belonging to Class B. We say that the functional module X includes the functional module Y if X is in a higher layer than Y and there is a top-dowm path from X to Y. The microprocessor in the top layer belongs to Class B, because most microprocessors can be designed by modifying the structures of registers, buses, ALUs, and so on, and redesigning the control circuits. Figure 4 shows a block diagram of a microprogram sequencer, which also belongs to Class B. Any two microprogram sequencers of different microprocessors are different at the logic gate level, even if they have the same block diagram. For example, a function set for branch address control such as a conditional jump differs from one processor to another, because the optimal function set depends on the processor architecture. Therefore, we must redesign and modify the microprogram sequencer at the logic gate level design, even if we use a previously designed block structure.

One of the most important considerations is the optimality of the whole processor, even though only small parts of it are modified. It is difficult to maintain the overall optimality, because we must modify many functional modules belonging to Class B simultaneously. However, even human designers cannot do this. They

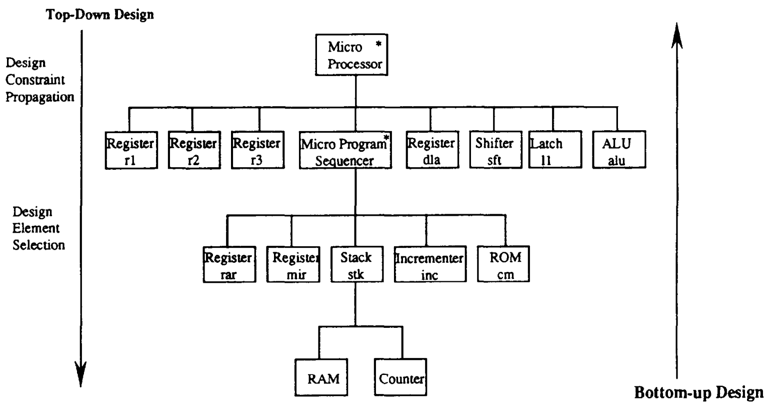

Fig. 3 Component structure of the microprocessor in Fig. 1.

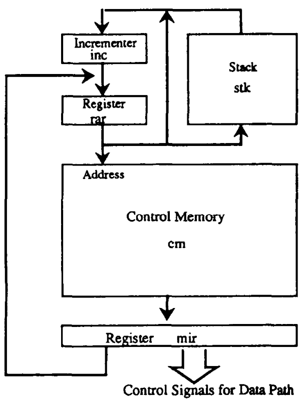

Fig. 4 Block diagram of the microsequencer.

search for the optimal solution for only one functional module at a time. Therefore, we assume that a quasi-optimal solution for the whole processor can be obtained if we optimally design the functional modules of Class B one by one. Namely, we assume the following.

Assumption 1

Suppose that the functional module Z includes the functional module Y of Class B. Then we can get a quasi-optimal solution of Z if we design Y optimally under the constraint that the specifications of Z and all the functional modules included in Y are satisfied.

When this assumption is introduced, even if we are designing a microprocessor that contains many functional modules belonging to Class B, the modules can be designed one by one. For example, in the microprocessor shown in Fig. 3, we can determine the detailed specifications of the microprogram sequencer so that the processor will be quasi-optimal while satisfying the detailed specifications of stack, registers, incrementer, and ROM, and the given specifications of the processor. Therefore, in designing the microprogram sequencer, we must finish synthesizing the functional modules in the layers lower than that of the sequencer, such as the stack, registers, incrementer, and ROM. Hence, when functional modules belonging to Class B are designed, we must use the bottom-up approach in which the functional modules in the lower layers are designed before those in the higher layers.

When the specifications of the micro-processor are given, the design process is carried out, first, by using the top-down approach. In this design step, the specifications of each functional module contained in the microprocessor are refined by constraint propagation and design element selection. We finish designing the functional module belonging to Class A by using only the top-down approach. Since the design data contained in the module library are utilized effectively in the top-down design approach, its search space is relatively narrow. On the other hand, after finishing the top-down design, we must carry out the bottom-up design for the functional modules belonging to Class B. In the bottom-up design approach, all the modules must be combined so that the specifications are satisfied. To do this, all module specifications must be analyzed. Therefore, the search space for the design becomes huge.

3. Modifying Functional Modules According to a Bottom-up Approach

Bottom-up design is an operation by which the specifications of functional modules belonging to Class B can be refined while taking account of all the specifications of the functional modules in lower layers. The Specifications of functional modules are classified according to the four kinds of descriptions shown in Table 1. They are discussed below.

3.1 Descriptions of Functional Module

1. Behavioral description

The behavioral description represents the behavior of

Table 1 Design Data Class and Descriptions.

Description \ Design Data Class	I	II	III	IV	V	VI
(1) Behavioral Description	×	O	×	O	×	×
(1)' Behavioral Description of Control Module	×	×	×	×	×	O
(2) Rough Structural Scheme Description	O	O	–	–	–	–
(3) Detailed Data Path Description	×	×	O	O	–	–
(4) Detailed Structural Description	×	×	×	×	O	O

× : not determined
O : determined
– : don't care

```
micro_operation_alu := {
        decode mir<0, 3> := {
                0 :=    dla = bus1 + bus2 +0;
                1 :=    dla = bus1 + bus2 +1;
                2 :=    dla = bus1 + bus2 +ct;
                3 :=    dla = bus1 & bus2;
                4 :=    dla = bus1 | bus2;
                5 :=    if (flag==1) dla = bus1 + bus2 +0;
                        else  dla = bus1 + bus2 +1;
               10 :=    if (flag==1) dla = bus1 & bus2;
                        else  dla = bus1 | bus2;
        }
}
```

Fig. 5 Behavioral description of the microprocessor in Fig. 1.

a functional module by using a programming language such as C. Figure 5 shows this for the microprocessor in Fig. 1. The description is of the execution function of the processor. According to this description, when the value of mir⟨0, 3⟩ is 0, the operation 'dla=bus1+bus2+0;' is executed, where mir⟨0, 3⟩ means the 0-th to third bits of the register mir. When the value of mir⟨0, 3⟩ is 1, the operation 'dla=bus1+bus2+1;' is executed.

2. Rough structural scheme description

The rough structural scheme description represents the rough scheme of a functional module structure with blocks and principal connections between them. Figures 1 and 4 are examples.

3. Detailed data path description

A functional module belonging to Class B is divided into two parts: a data path part and a control part. The former is a circuit that executes some functional operations by using functional modules such as ALUs, shifters, and registers. The latter is a circuit that supplies the functional modules of the data path part with control signals so that data processing can be executed correctly. Figure 6 shows an example of a detailed data path description corresponding to the microprogram sequencer shown in Fig. 4.

4. Detailed structural description

The detailed structural description represents the detailed structure of a functional module, which consists

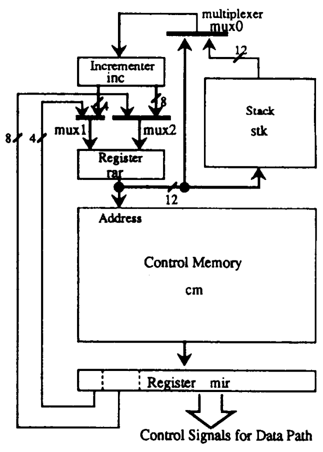

Fig. 6 Detailed structural description of the data path.

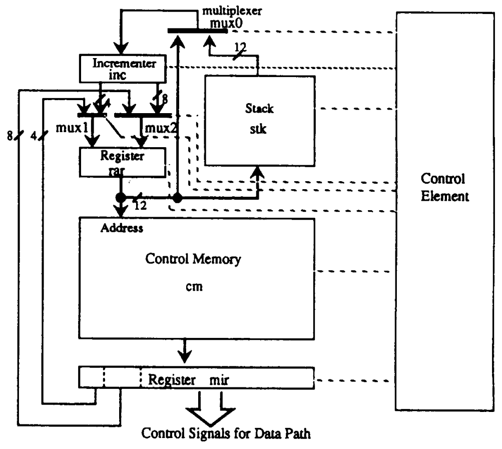

Fig. 7 Detailed structural description.

of a detailed data path structure, a detailed control structure, and their connections. Figure 7 shows an example of the detailed structural description corresponding to the microprogram sequencer shown in Fig. 4.

A functional module is represented by design data that are a combination of the above four kinds of descriptions. High-level synthesis is defined as a procedure for transforming abstract design data of the functional modules into detailed data. We define six kinds of design data to represent the same functional module at different abstract levels, as shown in Table 1. After the top-down design has been finished, design data for functional modules fall into one of these six categories. For example, when the behavioral description and rough structural scheme description are given as design data and the behavioral description of the control module is not given, the design data belong to Class II.

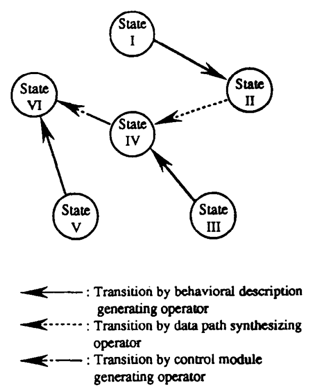
State I
State II
State III
State IV
State V
State VI
: Transition by behavioral description generating operator
: Transition by data path synthesizing operator
: Transition by control module generating operator

Fig. 8 State transition diagram for bottom-up design.

The design data belonging to Class VI are the most detailed. The design of functional modules described in this class is considered to be finished. Bottom-up design creates less detailed design data, belonging to classes I-V, on up to the most detailed, in class VI. Therefore, in order to implement the bottom-up design, we must implement the task that raises the design data of classes I-V to the most detailed data, belonging to class VI. We consider the design data belonging to one of the classes as a state. Therefore, the bottom-up process of designing functional modules is represented by a state transition diagram. Figure 8 shows a state transition diagram of the bottom-up design. From it, we can generate design data belonging to class VI from those belonging to one of the classes I-V, using three operators. In this paper we call them bottom-up operators.

These bottom-up operators are discussed in the following section.

3.2 Bottom-up Operators

1. Behavioral-description-generating operator

The behavioral-description-generating operator generates a behavioral description of a functional module belonging to Class B, analyzing both the specifications of all the functional modules in the lower layers and the given specifications of the microprocessor. For example, for the microprogram sequencer, the operator generates a behavioral description, analyzing the specifications of the ROM, incrementers, registers, stacks, and microprocessor. We assume that the following specifications are given for the microprogram execution sequence:

```
decode mir⟨0, 3⟩:={
    0:=rar=mir⟨0, 21⟩;                        (1)
    1:={stk(rar, 1); rar=mir⟨10, 21⟩;}        (2)
    3:=rar=stk(rar, 2)+1;                     (3)
    4:=rar=r3⟨0, 11⟩;                         (4)
}.
```

Both mir and rar are functional modules contained in the microprogram sequencer. Therefore, operation (1), which transfers data in mir to rar, can be executed in the micro-program sequencer. Operation (2) can also be executed in the sequencer, because all the functional modules used in (2), namely stk, rar, and mir, are elements of the sequencer. The operation +1, which appears in operation (3), can be executed in the incrementer 'inc' by looking into the specifications of 'inc'. The executability of the operation can be determined by looking into the specifications of the functional modules in the lower layers. Therefore, operation (3) can also be executed in the sequencer. The data in register '*r*3' are transferred into the register 'rar' by operation (4). However, register '*r*3' is not included in the microprogram sequencer. Therefore, operation (4) cannot be executed in the sequencer. Now, we assume that the output terminals of register '*r*3⟨0, 11⟩' are connected to the input terminals of the sequencer 'in⟨0, 11⟩'. The operation that transfers the data from 'in⟨0, 11⟩' to 'rar' can be executed in the sequencer, because all the modules used in the operation are contained in the sequencer. We can transform a non-executable operation into an executable one by looking into the specifications of the modules in the lower layers, and deciding the executability of the operation. We can obtain the following behavioral description of the sequencer by using the behavioral description generating operator:

```
decode mir⟨0, 3⟩:={
    0:=rar=mir⟨0, 21⟩;                        (1)
    1:={stk(rar, 1); rar=mir⟨10, 21⟩;}        (2)
    3:=rar=stk(rar, 2)+1;                     (3)
    4:=rar=in⟨0, 11⟩;                         (4)'
}.
```

2. Data-path-synthesizing operator

The data-path-synthesizing operator generates a detailed data path description. The data path generated by the operator executes data processing, which is described by the behavioral description. At first, the operator extracts the connections of the modules needed to execute the behavioral description by looking into the behavioral description of the functional modules and the specifications of all functional modules in the lower layers. It then transforms the connections into a detailed behavioral description by resolving data conflicts occurring with them.

Let us take the microprogram sequencer as an example. The operator extracts the connections by looking into the behavioral description of the sequencer and the specifications of the ROM, incrementer, registers and stack. The behavioral descriptions (1)-(4)′ shown in the "behavioral-description-generating operator" section are assumed to be given as a behavioral description of the sequencer. The operator extracts, from operation (1), the connection between the outputs of mir⟨10, 21⟩ and the inputs of rar. Similarly, from operation (2), it extracts the connections between the outputs of rar and the inputs of stk and between the outputs of mir⟨10, 21⟩ and the inputs of rar. From operation (3), it extracts the connections between the outputs of stk and the inputs of inc and between the outputs of rar and the inputs of stk. In operation (3), the operation '+1' appears. The connections for the operation '+1' are easily extracted by confirming that inc can execute the operation '+1'. In this way, the operator extracts the connections needed to execute the operations by looking into the behavioral description and the specifications of all functional modules in the lower layers.

Then, the operator transforms the connections into a detailed behavioral description by resolving data conflicts occurring with them. In order to execute operation (1), the connection between the outputs of mir⟨10, 21⟩ and the inputs of rar is needed. On the other hand, in order to execute operation (4)′, the connection between the inputs of the sequencer in⟨0, 11⟩ and the inputs of rar is needed. Therefore, data conflict occurs because two different data meet at the inputs of rar. The data conflict can be resolved by inserting a multiplexer to control the data flow at the point where the conflict occurs. The operator transforms the connection into a detailed data path description, as shown in Fig. 6, inserting multiplexers at the points where data conflicts occur.

3. Control-module-generating operator

The control-module-generating operator generates a detailed structural description. At first, the operator extracts the specifications of a control module. The control module is a functional module that controls data processing on the data path so that the behavioral description can be executed correctly. The operator extracts the specifications of the control module by looking into the behavioral description and the detailed data path descriptions of the functional module and the specifications of all functional modules in the lower layers. It then extracts the connections between the control module and all the functional modules contained in the data path. The operator generates a detailed structural description by merging both the connections and the control module into the detailed data path description.

As for the microprogram sequencer, the operator extracts the specifications of a control module and the connection between the module and the data path by looking into the behavioral description and the detailed data path description of the sequencer and the specifications of the ROM, incrementer, registers, and stack. We assume that the behavioral descriptions (1)-(4)′ shown in the section on the behavioral-description-generating operator are given as behavioral descriptions of the sequencer. In order to execute operation (1), the path from the outputs of mir⟨10, 21⟩ to the inputs of rar must be followed and the data along the path must be loaded into the rar. To do this, we need the control signals that cause the path to be followed and make the rar load the data along the path. The operator extracts the control signals needed to execute the behavioral description by looking into the specifications of all functional modules in the lower layers. Then, the operator generates a behavioral description of the control module and the connections between the control modules, ROM, incrementer, registers, and stack from the extracted control signals.

The state transition diagram shown in Fig. 8 is represented by a directed graph. Representing state transitions by inequalities, we obtain

$$\textit{State VI} < \textit{State IV} < \textit{State II} < \textit{State I}, \tag{1}$$

$$\textit{State VI} < \textit{State V}, \tag{2}$$

$$\textit{State VI} < \textit{State III}. \tag{3}$$

From the above inequalities, we can derive the order in which to apply the bottom-up operators as follows:

$$\begin{aligned}&\textit{Behavioral-description-generating operator} >\\ &\textit{Data-path-synthesizing operator} >\\ &\textit{Control-module-generating operator}.\end{aligned} \tag{5}$$

The bottom-up process of synthesizing a functional module is divided into six steps according to the determined design data of the module. The process is represented by using the state transition diagram, regarding each step as a state. The state transition is performed by using bottom-up operations. As shown in this section, the order in which to apply the bottom-up operators is easily derived by using the state transition diagram.

4. Implementation

New programming methodologies introducing the object-oriented paradigm have been studied extensively in recent years [10]. They aim at realizing an excellent programming environment making effective use of the following advantages of object-oriented programming:

1. The problem structure can be easily reflected in the program structure;
2. Higher modularity can be expected owing to the autonomy of objects;
3. Higher common usability and reusability of procedures can be expected as a result of the inheritance mechanism; and
4. Flexible controllability can be implemented owing to the dynamic control of procedures such as message passing.

These advantages of the object-oriented programming paradigm are especially useful in expert systems for design problems. Therefore, we built an expert system for high-level synthesis of VLSIs based on the object-oriented paradigm.

Figure 9 shows the system structure of the expert system, named ProLogic. It consists of four subsystems: a template object data base, a refinement operator data base, an instance object data base, and a task controller that controls the reasoning for design problems. These subsystems are discussed below.

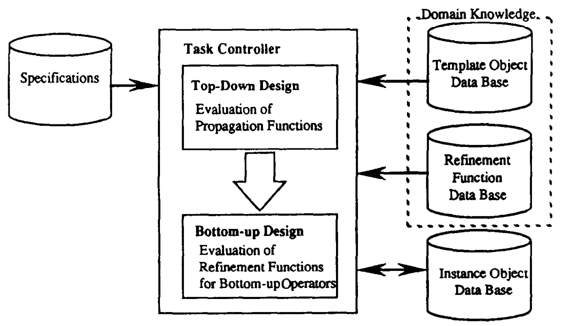
Domain Knowledge
Specifications
Task Controller
Top-Down Design
Evaluation of Propagation Functions
Template Object Data Base
Refinement Function Data Base
Bottom-up Design
Evaluation of Refinement Functions for Bottom-upOperators
Instance Object Data Base

Fig. 9 System structure of ProLogic.

```
micro_sequencer := {
    object_variables := {
        bit_length   : Bit
        delay        : D1
        timing       : T
        .......................
    }
    components_definition := {
        cm<Bit> := {
            type   : rom
            timing: next(1, T)
            delay  : mult(4, D1)
            }
        .......................
        }
    rough_structual_scheme_description := {
        rar[out] = cm[address_in];
        inc[out] = stk[in];
        rar[out] = inc[in];
        .......................
        }
    detailed_data_path_description := {
        data_path_synthesis
        }
    detailed_structural_description := {
        control_element_synthesis
        }
    behavioral_description := {
        repeat {
            mir = cm(rar);
            next
            extract(Bhv)
            }
        }
}
```

Fig. 10 Template object of the microsequencer in Fig. 4.

4.1 A Template Object and an Instance Object

In ProLogic, every functional module, such as a register, a counter, or a stack, is represented as an object. All the objects needed to describe functional modules are classified into two kinds, template objects and instance objects. The former are objects that represent only the concept of a functional module such as a counter or register. The concept of the functional module is a generic procedure or structure needed in synthesizing the module, and does not correspond to any entities of the functional module. On the other hand, instance objects represent the entity of the functional module; for example, when there is an object that represents a register with eight-bit width named '*r*1', the object represents the entity of '*r*1' and is an instance object. An instance object is generated according to the concept described in the template object. Figure 10 shows an example of a template object for the microprogram sequencer. In ProLogic, two mechanisms are provided in order to describe the concept of the functional module. One is an object variable and the other is a refinement function.

(1) Object variable

The object variable is described as a string of characters that is initiated with a large character similar to a Prolog variable. Bit, D1, and T, which appear in Fig. 10, are all object variables. Any values, such as numerics, lists, and atoms, can be assigned to an object variable. For example, we can assign [0, 10] to Bit and [1, *p*1] to T. When appropriate values have been assigned to all the object variables in the template, the names of objects are changed to those of instantiated functional modules such as '*r*1' and are copied. The copied objects are instance objects generated by the template objects.

(2) Refinement function

In ProLogic, a function that refines specifications can be defined, and is called a refinement function. Examples are the functions mult (4, D1) and next (1, T), which appear in Fig. 10. The bottom-up operators defined in Section 3.2 are also implemented as refinement functions. The behavioral-description-generating operator is implemented by using the refinement function 'extract (Bhv)', the data-path-synthesizing operator by using 'data_path_synthesis' and the control module generating function by using 'control_element_synthesis'. The specifications of the functional modules can be refined by evaluating the refinement functions. When all the refinement functions in the object have been completely evaluated, the design process for the module described by the object is finished. The refinement function procedure is defined in the refinement operator data base and is described in an ordinary programming language such as FORTRAN, C, or Prolog.

The evaluation priority of the refinement functions is controlled by the task controller as described next.

4.2 Task Controller

In ProLogic, the refinement functions used in the top-down design approach can be explicitly disitinguished from those used in the bottom-up design approach. As we mentioned in Section 2, the design process consists of two steps. In the first step, the specifications of the functional modules are refined by using the top-down approach. In the second step, the bottom-up approach is taken. Therefore, we must control the reasoning so that the refinement functions used in the top-down approach can be executed first and then those used in the bottom-up approach can be executed.

1. Control method for evaluation of refinement functions used in the top-down approach

The process of top-down design consists of design constraint propagation and design element selection. Design constraint propagation can be executed by assigning values to object variables and evaluating the refinement functions used in the top-down approach. These functions are called propagation functions. In the template object shown in Fig. 10, next (1, T), mult (4, D1), and add (1, B) are all propagation functions.

The procedure for evaluating a propagation function is defined in the refinement operator data base. The task controller can control the evaluation priority of propagation functions. This priority can be defined explicitly as follows.

```
1 mult
2 add
3 next
.......
```

In the design constraint propagation phase, the propagation functions are evaluated in this order.

As shown in Fig. 10, all functional modules in the lower hierarchy are defined in the component_definition slot of a template object. The design constraints can be propagated via this slot. For example, when the object variable T is assigned to [1, *p*1], the start timing of cm is determined to be next (1, [1, *p*1]). Then, the propagation function next (1, [1, *p*1]) is evaluated and determined to be [1, *p*2]. In this way, we can determine the components_definition by means of refinement function evaluation and object variable assignment.

2. Control method for evaluation of refinement functions used in the bottom-up approach

There are three bottom-up operators in ProLogic, as mentioned in Section 3.2. The bottom-up operators are implemented as refinement functions used in the bottom-up design approach. The refinement functions used in the bottom-up approach should be defined by taking account of their orders of application, shown in inequality (4). The evaluation order of the refinement functions of the bottom-up operators can be defined explicitly as follows:

```
1 extract
2 data_path_synthesis
3 control_element_synthesis
```

In the bottom-up design approach, the refinement functions corresponding to the bottom-up operators are evaluated in the defined order.

5. Execution Results and Their Evaluation

We implemented ProLogic in the logic programming language Prolog, and synthesized a microprocessor that contains about 36,000 transistors by using ProLogic. ProLogic synthesized the logic circuits according to the design strategies described in the template objects. Therefore, the synthesized block structures of the microprocessor in all layers are the same as those designed by human experts. The number of transistors synthesized by ProLogic is less than 105% of the number designed by human experts. The quality of hardware synthesized by ProLogic is almost the same as that of hardware synthesized by human experts in terms of the block structure and quantity of hardware, which are the two most important parameters determining quality. This confirms that the reasoning method discussed is suitable for building expert systems for logic circuit design.

The knowledge base of ProLogic consists of the refinement function data base and the template object data base. In an ordinary expert system, domain knowledge for the specialized problem is registered in the knowledge base. Therefore, a different expert system can be built by replacing the knowledge base. With ProLogic, it is also possible to build another expert system by exchanging the refinement functions and template objects. The reasoning discussed in this paper controls the design process, which is divided into two steps, the top-down design approach and the bottom-up design approach. The top-down design approach is suitable for design problems in which the design elements are refined according to experience and knowledge of design. On the other hand, the bottom-up design approach is suitable for design problems in which the design elements are synthesized and combined correctly by analyzing the specifications. We can adapt the reasoning mechanism to various design problems. As an example, we considered a software design problem. When software is designed, its function is decomposed into many sub-functions according to experience and knowledge about software design. Each sub-function is refined as a program module. Then, each module is synthesized and a program is generated by using control statements such as *if* and *while*. The procedure for decomposing the software into sub-functions can be implemented by using the top-down design approach. The procedure for program module synthesis can be implemented by using the bottom-up approach.

6. Conclusion

We have discussed a new methodology for building an expert system for high-level synthesis of VLSIs. We found that design problems could be decomposed into two steps: top-down design, which is known as routine design, and bottom-up design. We therefore developed a reasoning mechanism that can control both the top-down and bottom-up design approaches.

We built an expert system that can execute the proposed reasoning. Then, we showed that the reasoning mechanism can be easily implemented by introducing an object-oriented programming paradigm and using Prolog. An expert system for high-level synthesis, named ProLogic, was built according to the proposed reasoning. The block structure of the logic circuits synthesized by ProLogic was confirmed to be as good as that designed by human experts. The total amount of hardware synthesized by ProLogic is less than 105% of that specified by humans. Therefore, the quality of the logic circuits is considered to be as good as that of circuits designed by human experts.

Acknowledgements

We would like to express our sincere thanks to Dr. Yukio Kawamoto and Mr. Masao Yanaka of Hitachi Ltd. for giving us the opportunity to carry out this research project. We would also like to thank Dr. Hideo Maejima, Mr. Koyo Katsura, and Mr. Shigeru Matsuo of HRL for many helpful discussions regarding the CMOS VLSI logic design.

References

1. Thomas, D. et al. Automatic Data Path Synthesis: *IEEE, Comput.*, **16** (Dec. 1983), 59-70.
2. Kowalski, T. J. et al. The VLSI Design Automation Assistant: From Algorithms to Silicon, *IEEE, Design & Test*, **2** (Aug. 1985), 33-43.
3. Kowalski, T. J. et al. *An Artificial Intelligence Approach to VLSI Design*: Kluwer Academic Publishers, 1986.
4. Yokota, T. et al. A VLSI Design Automation System Using Frames and Logic Programming: *IEEE, Proc. 3rd CAIA* (Feb. 1987), 296-301.
5. Hamada, N. et al. Expert Systems for VLSI Design, *Hitachi Review*, **37**, 5 (1988), 339-344.
6. Bekki, K. et al. A Method for Microprogramming Control Logic Synthesis: *Trans. IPS Japan*, **29** (1988) 605-613.
7. Chandrasekaran, B. Generic Task in Knowledge-Based Reasoning: High-Level Building Blocks for Expert System Design, *IEEE Expert* (1986), 23-30.
8. Chandrasekaran, B. Towards a Functional Architecture for Intelligence Based on Generic Information Processing Tasks: *Proc. IJCAI-87* (Aug. 1987), 1183-1192.
9. Brown, D. and Chandrasekaran, B. Knowledge and Control for a Mechanical Design Expert System: *IEEE Comput.*, **19**-7 (1986), 92-100.
10. Kamimura, T. Object-Oriented Extensions of Procedural Languages; *J. IPS Japan*, **29**, 4 (1988), 310-317.