Using the temporal logic programming language Tokio for algorithm description and automatic CMOS gate array synthesis

Masahiro Fujita*, Makoto Ishisone, Hiroshi Nakamura, Hidehiko Tanaka, and Tohru Moto-oka

Faculty of engineering, University of Tokyo
7-3-1 Hongo Bunkyo_ku Tokyo, Japan 113

*: Now at FUJITSU LABORATORIES LTD.

Abstract

To date, simulation has been the primary method used to support hardware logic design. In particular, there has been little that could support such design from the system level, such as a language to describe processing algorithms. In this paper, we will propose a silicon compiler, which is designed to support CMOS gate arrays from the system level. Descriptions from the system level through the state diagram level are done by using the temporal logic programming language called Tokio. Being based on temporal logic, Tokio enables the use of temporal operators. This facilitates the description of concurrent operations that cannot be easily described in Prolog. In addition, because the mathematical models are clearly defined, verification and synthesis can be easily supported. At present, only a simulator coded in Prolog is available to support descriptions of this level in Tokio.

A program that automatically synthesizes logical circuits for CMOS gate array from state diagram level is supported. The core of this program has already been developed in C-Prolog. Synthesis has already been tested at our laboratory. We used the Unify Processor (UP) of PIE (Parallel Inference Machine) which is currently under development, as an example.

This paper introduces the hardware design support strategy based on Tokio, and explains the details of the program that synthesizes CMOS gate arrays from the descriptions of the state diagram level.

1. Introduction

Significant progress in device and implementation technologies has made it possible to create large-scale, complicated hardware systems. However, conventional CAD technology is primarily related to the implementation process. This means that conventional CAD technology is not powerful enough to support hardware logic design at a higher level. A need has developed for a hardware logic design support system, also known as a silicon complier, that can support logic design from the system level, and that can be liked to an implementation design CAD system.

For the language to be used in the hardware logic design support system, we proposed a logic programming language, which has enough mathematical backgrounds. A representable logic programming language is Prolog. The classical predicate calculus on which Prolog is based, however, does not incorporate the "time" concept. In other words, it does not allow descriptions of concurrent operations, and it requires special techniques for hardware descriptions. Temporal logic (Manna and Pnueli 1981, Moszkowski 1983) includes the "time" concept, and Tokio (Aoyagi et al. 1985, Kono et al. 1985) - a programming language

based on temporal logic - was selected to describe hardware. The proposed design support system will provide the verification and synthesis capabilities for design descriptions written in Tokio, based on the theory of temporal logic.

The proposed system, also known as a silicon complier, will receive algorithm descriptions written in Tokio, execute the design process, and finally generate the CMOS gate circuit. CMOS gate arrays are selected, because they are provided with a relatively high degree of integration and a sophisticated CAD system for implementation.

2. Temporal logic programming language Tokio (Aoyagi et al. 1985, Kono et al. 1985)

Temporal logic is considered an ordinary classical logic that is additionally supplied with several temporal operators to enable time-related description. Such logic is defined in terms of discrete time, and temporal operators can be used to specify a value for each variable at each time. Temporal logic such as Linear Time Temporal Logic (LTTL) (Manna and Pnueli 1981) and Interval Temporal Logic (ITL) (Moszkowski 1983) are available, depending on the operators provided. Tokio is an extended version of LTTL, uses the easy-to-describe characteristics of ITL, and can be easily converted into LTTL. LTTL is decidable in the range of propositional logic, and research into this logic has made substantial progress (Manna and Pnueli 1981).

Just as Prolog is a logic programming language based on classical logic, Tokio is a temporal logic programming language based on temporal logic. Because temporal logic includes classical logic, Tokio includes Prolog. That is, Tokio is a version of Prolog that has been extended to describe concurrent processing. The use of Tokio, therefore, contributes to simplifying hardware descriptions at different levels, from the system level through the gate level. The following discusses the configuration of the proposed hardware design support system in which Tokio serves as the nucleus.

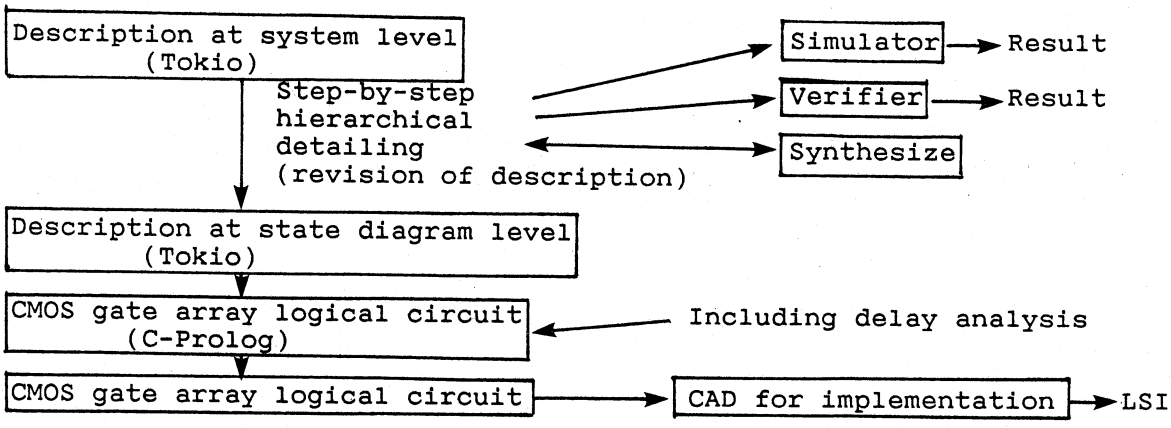


Fig. 1 Configuration of hardware design support system based on temporal logic programming language Tokio

3. Hardware design support system based on Tokio

Figure 1 shows the configuration of the proposed hardware design support system. First, the designer must determine the processing algorithm. Specifically, the designer must describe the algorithm in Tokio, and verify it by using a simulator and a verifier.

This algorithm description may not physically match the hardware.

Therefore, the designer must continue to revise the design description until it conforms to the actual hardware environment. During this process, the design si detailed step-by-step to match the processing image. In other words, it is described in a hierarchical manner. The following functions support this design description detailing procedure:

(1) Simulation
(2) Verifying whether the two given Tokio descriptions are identical
(3) Synthesizing a Tokio description with physical entity (data path) information to generate a more detailed Tokio description

This design detailing procedure is carried out up to the state diagram level (i.e., the same level as hardware description language DDL (Juley and Dietmeyer 1968)). When the state diagram level is reached, the Tokio description is expanded into a state transition table. Then, it is processed by the CMOS gate array logical circuit synthesis program. This program has been completed for the most part, and is discussed in the next chapter. This program outputs a logical circuit description for CMOS gate arrays. This output is forwarded to the implementation design CAD, where it is used to design an LSI gate array.

4. CMOS gate array logical circuit synthesis program

The system configuration shown in fig. 1 indicates that the CMOS gate array logical circuit synthesis program receives the Tokio description at the state transition diagram level. However, because Tokio is not yet fully supported, the program currently receives DDL (Juley and Dietmeyer 1968). Tokio, as explained in chapter 2, describes intervals that consists of several consecutive time series each. DDL is considered to consist of such intervals whose lengths are all 1. In other words, DDL can be considered the result of detailing a Tokio description down to the state diagram level.

The CMOS gate array logical circuit synthesis program has the DDL translator (Juley and Dietmeyer 1969) to process the received DDL description. The program then receives the translator output. The DDL translator output includes the following:

- Terminal transfer table
- Register (including memory) transfer table
- State transition table
- Table of logical expressions for dummy terminal (generated by the translator and composed of conditional expressions for transfer and transition)
- Cross-reference table

The transfer table shows the transfer range for each destination terminal register (bits to be transferred or, for memory, addresses), source of transfer, and transfer conditions. The state transition table indicates the current state, next state, and transition conditions. Fig. 2 is an example of the register transfer table.

MAR (0 : 9)		10 BIT REGISTER
SINK RANGE	SOURCE	TRANSFER CONDITION
(0 : 9)	IAR (0 : 9) ADR (0 : 9)	ADS DEC

Fig.2 DDL translator output example (register transfer table)

4.1 Logic design of CMOS gate arrays

In a CMOS gate array, basic gates called "basic cells" are arranged on the chip. Each basic cell consists of several n-MOS and p-MOS transistors. Fig. 3 shows equivalent circuit of two n-MOS and two p-MOS transistors. The desired circuit can be obtained by connecting these basic gates by aluminum wires.

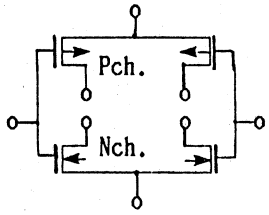


Fig. 3 Basic cell equivalent circuit

Basic cells do not function as logical elements until they are wire-connected. The manufacturer, therefore, first determines the basic logical element wiring modes to be used in logical circuit design. These modes include NOT, NAND, flip-flop, etc., which are registered in a library. The collection of logical elements registered in this library is called the unit cell family, and the user designs their logical circuit by using elements in the family.

4.2 Flow of circuit synthesis process

The CMOS gate array logical circuit synthesis program is expected to input DDL translator output (terminal and register transfer table, state transition table, etc.), and generate a logical circuit at the unit cell level for CMOS gate arrays. At this time, the program must perform the following three operations:

(1) Simplifying the circuit
DDL translator output such as transfer table, has a relatively large redundancy. This means that, if such output is expanded into unit cell circuit as is, the number of required gates and delay time will both be significantly increased. In addition, the circuit to be generated through the synthesis process should match the characteristics (based on NAND and NOR) of the CMOS gate array logical circuit by considering the number of gates and the delay time. Therefore, the circuit must be simplified during the circuit synthesis process.

(2) Fan-in/fan-out considerations
The maximum fan-in for unit cells is limited, meaning that an element with a fan-in value exceeding the maximum must be divided into several unit cells. The maximum fan-out is also limited by the manufacturer, and the circuit to be generated must satisfy the maximum fan-out value.

(3) Analyzing the number of gates and delay time
The maximum number of gates is determined by the gate array to be used. The delay time determined by the clock speed. If the generated circuit exceeds these maximum values, logic separation or design modification is necessary.

For such complicated processing, it is not efficient to directly synthesize circuits at the unit cell level. To solve this problem, the synthesis program uses virtual units called "macro units". In other words, the circuits are first synthesized at macro unit level. Macro

units, as virtual logical elements, have the following characteristics:

(1) Unlimited fan-in
(2) Infinite fan-out
(3) Input N bits and output N bits
(4) Functionally more advanced (more abstract) than unit cells

For example, in addition to macro units that perform basic operations such as AND and OR, there are those that process more advanced function like addition/subtraction, comparison, and register handling.

When compared to unit-cell-level circuit, the use of macro-unit-cell circuit for simplification or gate count and delay analysis will result in the following advantages:

(1) Less data needs to be manipulated, which simplifies and speeds up processing.
(2) Because its functions are more abstract, it can be technology independent to some extent. This means that this system can also be logically applied to TTL circuits by modifying the operations following the macro unit level.

We decided, therefore, that the synthesis program should read the DDL translator output, expand it into macro units, and execute the various operations for expansion into unit cells.

4.3 Outline of processing

The synthesis process is divided into six phases according to the decisions described above (see table 1).

Phase1: Converts the DDL translator output to Prolog description
Phase2: Expands Prolog description into macro units, performs simplification 1, and generates a cross-reference table
Phase3: Performs simplification 2
Phase4: Analyzez the number of gates and delay time, and modifies the design according to the analysis results
Phase5: Expands macro units into unit cells and performs simplification 3
Phase6: Converts expansion result data for tools

Table 1 Process phases

In phase 1, the program receives the DDL translator output including terminal and register transfer tables, a state transition table, and a table containing logical expressions for transition and transfer conditions. The program converts this output into Prolog description format.

In phase 2, the program expands the data obtained in phase 1 into a circuit for which macro unit are used. At the same time, it performs simplification 1 and generates a cross-reference table. Simplification will be explained in detail in the next section. The result of the expansion in this phase include such general function as AND, OR, and REGISTER, and are not dependent on any specific technology.

In phase 3, simplification 2 is performed at the macro unit level. The circuit is converted into one suitable for CMOS gate arrays. Details on this conversion will be given later.

In phase 4, the program analyzes the number of gates and the delay

time. Macro units are considered to have an infinite fan-out as previously explained. During this process, however, fan-out analysis and buffer insertion are also done by taking into account the operation used to expand macro units into unit cells. The values obtained in phase 4 are not fully accurate because the precise delay time cannot be determined until the cells are mounted on the chip. In addition, the number of gates will be changed by the simplification to be performed in phase 5.

In phase 5, the program expands the macro units into unit cells. Simplification 3 is performed at the same time.

In phase 6, the program converts the obtained unit cell circuit data so that it can be input to various tools, such as the simulator, for more precise delay analysis.

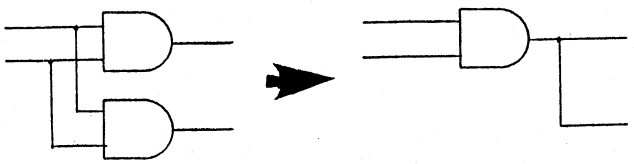
Simplification

As described above, there are three levels of simplification; number 1, 2, and 3, corresponding to the logical expression, macro unit, and unit cell levels, respectively. The following operations are done during there three simplification procedures:

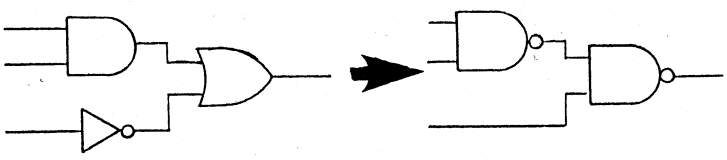
- Simplification 1: Fetches common parts from similar logical expressions (see (a) in fig. 4).
- Simplification 2: Primary simplification. Eliminates duplicated units (see (b) in fig. 4) and performs replacement according to the specific rules for optimization (see (c) in fig. 4). At this time, simplification according to the CMOS gate array characteristics, i.e., using NAND/NOR for basic gates, is performed.
- Simplification 3: Optimizes the constant and data calculator (see (d) in fig. 4).

```
T1 = A & B & C & D                COM = A & B & C
T2 = A & B & C & E      =>        T1 = COM & D
                                  T2 = COM & E
```

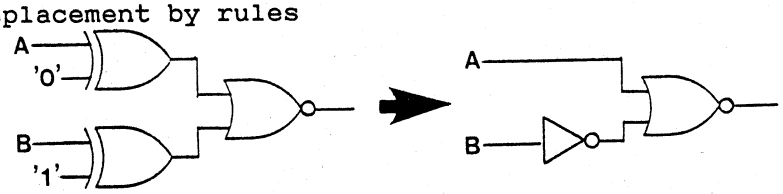
(a) Combining common parts (& represents AND)



(b) Eliminating duplicate gates



(c) Replacement by rules



(d) Optimizing constants

Fig. 4 Implementation by Prolog

4.4 Implementation by Prolog

The system proposed in this paper is written in C-Prolog [9], which is developed at Edinburgh university. The simplification operations, which are important factors in implementations, are discussed below.

Simplification 1

During simplification 1, which is performed at the logical expression level, common portions of similar logical expressions are unified. The algorithm is as follows.

- Reads one logical expression
- Fetches the common part of previous expressions, which is stored in a separate area, and merges it with the newly obtained expression to generates a new common part.

IF a common part does not exist,
THEN
- Expands the previous common part into a macro unit
- Stores the newly obtained expression as the new initial value of the common part

ELSE
- Stores the new common part and obtained expression as data

ENDIF
- Repeats the above operation for all expressions. When there are no more expressions:
  IF any common part and expression are still stored
  THEN
  - Expands them into macro units.

Simplification 2

This is the primary simplification of the three simplification steps. IT is further divided into four as follows:

(1) Unification of units that have the same functions, same inputs, but different nets (see figure 5)
(2) Simplification of AND or OR gates that have several identical inputs (see figure 6)
(3) Simplification of multiplexer gates containing sets that have the same source under different conditions (see figure 7)
(4) Simplification according to the replacement rules

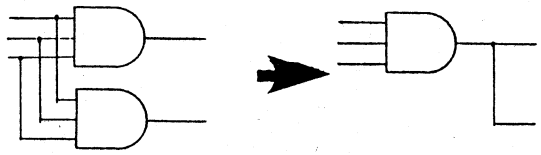


Fig. 5 Simplifying gates with different outputs

Fig. 6 Simplifying gates with identical inputs

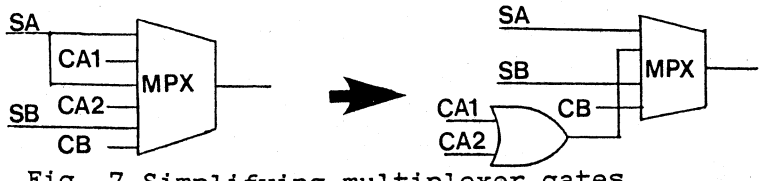


Fig. 7 Simplifying multiplexer gates

Fig. 7 Simplifying multiplexer gates

In stage (4), the program finds circuit patterns to which the replacement rules are applicable, and replaces the units following the rules. This results in:
- Dividing multi-input AND and OR gates
- Optimizing NOT-NOT and AND-OR gates

Thus, the stage 4 procedure consists of two major operations:
- Finding circuit patterns to which replacement rules apply
- Replacing them according to the rules

The following shows the definition of predicate "optimize" generated for the above.

```
optimize(macro(F,ID,B,IO)):-
        match(N,STAT,ID),
        substitution(N,ID,Info),
        (
                STAT == opt,
                count_info(Info,M),
                M < 0
        ;
                STAT == opt
        ),
        exec_info(Info),
        !.
```

The following explains the predicates used in this definition.

match(N,STAT,ID)
If the unit with the specified identification number (ID) matches the pattern determined by a rule, this returns the rule number (N) and rule type (STAT).

substitution(N,ID,Info)
This applies the rule with the specified number (N) to the unit with the specified identification number (ID), and returns the replacement data (Info). Note that it does not perform any actual replacement.

count_info(Info,M)
This returns the amount by which the number of gates will change (M) when the specified data (Info) is executed.

exec_info(Info)
This actually replaces the unit according to the specified data (Info).

In other words, the "optimize" operation can be expressed as "Find a unit that matches the pattern of a rule. If the rule is not of the optimization type (STAT=/=opt), then replace the unit; if the rule is of the optimization type, then only replace the unit if the number of gates will descrease (M<0)."

4.5 Example of synthesis: Unify Processor (UP)

For an example of this synthesis system, we used the Unify Processor (UP) of the highly parallel inference engine PIE (Goto et al. 1983), which is currently being developed, for conversion into CMOS gate array circuits.

The UP currently used in PIE was designed manually. It consists of

approximately 500 TTL IC circuits and 17 internal registers (356 bits in total), and is controlled by a microprogram. For this example, we described the UP in DDL, converted it through the DDL translator as the data to be processed. The source UP description written in DDL was approximately 1000 lines long.

Result of synthesis

Process times

Table 2 lists the process times required for individual phases. These values are CPU times measured on a VAX11/730.

Phase1	5 h 30 m
Phase2	
Expansion	40 m
Cross-reference	11 h 30 m
Phase3	74 h 30 m
Phase4	
Fan-out analysis	3 h 30 m
Gate count analysis	10 m
Delay time analysis	2 h
Total	92 h

Table 2 Process time

Results of expansion into macro units

Table 3 shows the results of expression into the macro units at the end of phase 2.

	Macro unit count	Basic cell count
State transition	43	124
Registers	19	4752
Memory	8	0
Memory address	23	1380
Register transfer	344	7646
Terminal transfer	122	5041
Dummy terminal	1796	7600
Total	2355	26543

Table 3 Result of expansion into macro units

Effect of simplification (Table 4)

	Gate count	Change	Required time
Initial	26543		
Simp 1	19666	-6877	6 h 30 m
Simp 2	17467	-2199	1 h
Simp 3	16604	-863	17 h 40 m
Simp 3	16405	-199	8 h

Table 5 Effects of simplification

As shown above, the number of gates has been reduced by 10,000, which is approximately 40 % of the total. Simplification 3 (Simp 3) was preformed twice because there may be remaining circuit patterns to which the rules of replacement apply to at the end of first Simp 3. Simp 3 was not executed more than twice because the number of gates did not significantly change, and because the execution time should be minimized.

5. Conclusion

In this paper, we described the hardware logic design support system based on temporal logic programming language Tokio and the automatic CMOS gate array synthesis system. At present, Tokio can only be supported by the interpreter written in C-Prolog, but we are now creating a high-speed processing system based on the C language. We are also discussing a new verification/synthesis system.

The nucleus of the automatic CMOS gate array synthesis system has been completed using C-Prolog. We have shown that the system can synthesize approximately 20,000 gates, and the execution time can be reduced to several hours by using a large-scale computer.

We are now working to facilitate greater support of Tokio to create a real silicon compiler.

References

Manna Z, Pnueli A (1981) Verification of concurrent programs part I: the temporal framework, Stanford univ. rep. STAN-C81-836

Moszkowski B (1983) Reasoning about digital circuits, Stanford univ. rep. STAN-CS-83-970

Aoyagi T, Kono S, Fujita M, Moto-oka T (1985) Logic Programming Conference '85, Tokyo Japan

Kono S, Aoyagi T, Fujita M, Tanaka H (1985) Logic Programming Conference '85, Tokyo Japan

Fujita M (1984) Logic design assistance with temporal logic, phD dissertation, Univ. of Tokyo

Juley JR, Dietmeyer DL (1968) A digital system design language (DDL), IEEE trans. computer, vol. C-17, no. 9

Juley JR, Dietmeyer DL (1969) Translation of a DDL digital system specification to equation, IEEE trans. computer, vol. C-18, no. 4

Pereira F (1984) C-Prolog users manual version 1.5, EdCAD, Edinburgh univ.

Goto A, Aida A, Tanaka H, Moto-oka T (1983) Highly parallel inference engine PIE, Logic Programming Conference '83, Tokyo Japan