Retargetable Netlists Generation and Structural Synthesis based on A Meta Hardware Description Language : Melasy+

Sho Nishida
Graduate School of Science and Technology
Shinshu University
4-17-1 Wakasato, Nagano, Japan
11ta541a@shinshu-u.ac.jp

Katsumi Wasaki
Faculty of Engineering, Shinshu University
4-17-1 Wakasato, Nagano, Japan
wasaki@cs.shinshu-u.ac.jp

***Abstract*—We are developing a compiler system, Melasy+, which is at a level higher than those of various model checking and hardware description languages. Melasy+ describes a single code and allows model checking and operation tests on an actual machine via a code generator for each language. In this study, for an XML intermediate representation code which was output by Melasy+, the elements which consists the target circuit are analyzed to generate a detailed list and then static analysis of the circuit is carried out. The Netlist after regeneration is a digraph and the meta-information obtained at the analysis is given to its edge. By exploring the digraph, a static analysis function of to detect structural error and/or redundant part, such as unused portion of circuit, asynchronous loop structure, register-register critical path, can be realized.**

I. Introduction

Electronic devices operate in a variety of environments due to the progress of IT technologies. Many social systems depend on these technologies and their reliabilities and highly reliable hardware is essential to maintain the social systems. To maintain the reliability of hardware, it is necessary to have functions such as integrated self test and there is a tendency of the increase in the scale and complexity of circuit. In hardware design, therefore, an environment is required where high reliability of both the required circuit itself and the self test circuit are secured and design / verification can be carried out efficiently [1][2]. Compilers to obtain objects and executable modules for a target system through code generation by implementing the target system [3][4] were developed. In recent years, hardware compilers [5][6][7][8] have been used which generate circuit configuration information directly from the code written in a relatively high-level language to describe the design of the hardware. There are model checking tools [9][10][11] such as NuSMV [12] to check the validity of hardware design as a matter of form. The validity of design can be evaluated automatically by using these tools. However, to carry out model check using NuSMV, it is necessary to use a very low-level language and a process is required to newly describe the system for verification purpose which was designed in a language such as VHDL [13] or Verilog [14] using NuSMV. Describing the system in the same hardware several times in different languages is difficult in terms of consistency in design and it is also inefficient from the viewpoint of process management. To solve these problems, a development environment was required in which a system can be described clearly and the tasks from design, verification to implementation can be carried out consistently. We have been developed a meta hardware description language Melasy+ [15] with a purpose to automatically carry out code generation for various existing languages, such as hardware description languages and languages for model checking, and to carry out the tasks from verification to implementation consistently. Melasy+ of the previous studies does not have a function to check the behavior and description of hardware described as Melasy+ code by itself. Therefore, to check errors in the design, it was necessary to go back to the final code generation for various processing systems. In this study, we suggest the enhanced functions for checking the description of Melasy+ code in an earlier stage and analyzing the described circuit in the Melasy+ environment. The check and analysis of the description are realized by regenerating a detailed Netlist based on an XML intermediate file used for code generation for various processing systems and by scanning on the Netlist. By carrying out the analysis of the Netlist, the check of components generated by the described Melasy+, detection of asynchronous loop structures, count of logic gate stages and the like can be carried out. Using the results of this study, the analysis of description, which should be carried out by another processing system in the conventional environment, can be carried out with Melasy+ only. By enhancing the analysis function to Melasy+ itself, it is possible to promote the improvement of upper-level design at the stage earlier than that in the conventional environment.

II. Meta Hardware Description Language: Melasy+

A. Overview

Melasy+ is a processing system at a level higher than that of existing processing systems such as NuSMV and VHDL. Melasy+ consists of an intermediate code generator

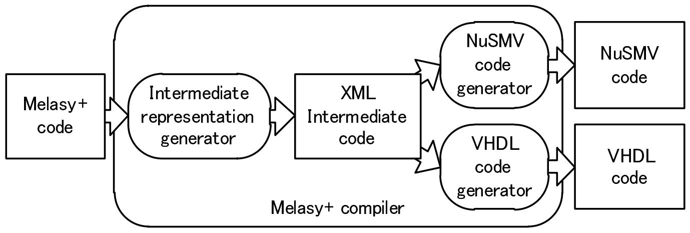


Fig. 1. Positioning of Melasy+ compiler and code generation

and code generators for various processing systems (Figure 1). The intermediate generator is implemented as a C++ class library and the Melasy+ code gives a description using the language functions of C++. By providing the Melasy+ code to the intermediate code generator, an intermediate code is obtained. By providing the obtained intermediate code to the code generators for various processing systems, codes for the languages are obtained.

B. *Language Functions*

The Melasy+ code gives a description as a class enhancement of C++ and uses the syntax and functions of C++. However, the standard types such as int and char cannot be used to hold values in a description by Melasy+. Instead, a Logic type or Digit type is used to hold values in Melasy+. The Logic type holds a value of 1 (one) bit and represents a binary state of low or high. The Digit type represents a value of fixed-length bit and can specify a bit width to a template argument. To substitute a value, int type or const char type is used. When describing hardware, Melasy+ describes a functional part by component to configure the whole hardware with the connections and hierarchical structure of the components. The description of a component is carried out using the function of the Component class. Input and output are defined using the functions such as in, out and sync. The functions such as switch, case and default are used for the definition of outputs and special conditional branching syntax can be used. It is also possible to use the language functions of C++ such as for and if. The defined component is called as a function and is used after the instantiation of it. For the reusable parts in a circuit description, the codes can be reused by calling the same definitions.

C. *Generation of XML Intermediate Code*

An executable file can be obtained by compiling the Melasy+ code to which a C++ library file, or an intermediate code generator, is included. By executing the executable file obtained, code generation is carried out to obtain an intermediate representation in the XML format.

D. *Code Generators for Various Processing Systems*

Now two processing systems, NuSMV and VHDL, are compatible with the code generation. Both code generators are described using Python and their inputs are the XML intermediate code of Melasy+. The code generator for NuSMV generates a .smv file and the code generator for VHDL generates a .vhd code.

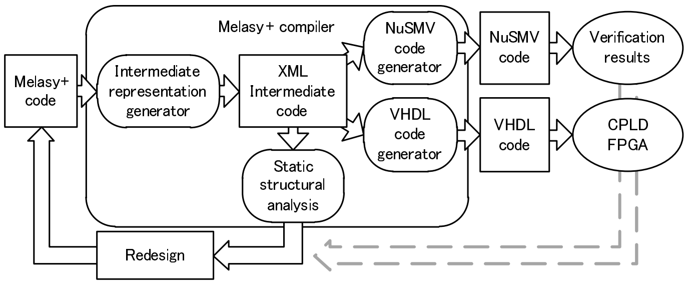


Fig. 2. Design cycle using generated codes or intermediate codes

III. Generation of Netlist

A. *Reconstruction of Netlist from XML Intermediate Code*

Although the intermediate representation code described in the XML format of Melasy+ contains the information to configure a circuit, the information is not specific. Therefore, it is not suitable for structure analysis and the like. In this study, with a purpose to carry out an analysis of the hardware design described by the Melasy+ code, a Netlist is regenerated from the intermediate representation code in the XML format. By reproducing a structure on software after providing a specific circuit structure to the Netlist to be regenerated, exploration of circuit structure in structure analysis and the like is made possible. The purpose is to improve upper-level design at the stage earlier than that of the conventional studies (Figure 2), based on the information obtained through the implementation of structure analysis on the Netlist. A Netlist generator is implemented as a C++ library file. It consists of a definition of the class configuring the Netlist and a parser whose input is an XML intermediate representation code. The Node class can hold the meta-information which can be obtained when the Netlist is analyzed in addition to the information representing the circuit configuration elements. The circuit structure on the Netlist is comprised of a series of instances of the Node class representing I/O ports and logic gates. The XML intermediate representation code is parsed to generate Node instances which correspond to the circuit configuration elements appeared. The circuit components can be represented through instantiation by providing the information of the circuit configuration elements to the Node class. The generated Node instances are connected based on the wiring information of the XLM intermediate representation code. The circuit structure is reproduced by generating the Node instances for all the circuit configuration elements described in the XML intermediate code and by connecting them. The DTD definitions defining the descriptions which may appear in the XML intermediate representation code are shown in Figure 3 in the form of source code.

B. *Circuit Configuration Management Table*

When generating Node instances, a Netlist is realized by reflecting the information of the circuit components formed by the Node instances to a circuit configuration management

```
<!ELEMENT melasy (component+)>
<!ELEMENT component (in | out | instance)*>
<!ATTLIST component name CDATA #REQUIRED>
<!ELEMENT in EMPTY>
<!ATTLIST in name CDATA #REQUIRED>
<!ATTLIST in type CDATA #REQUIRED>
<!ENTITY % expr "(op | op1 | port | const | switch)">
<!ELEMENT out (%expr;)>
<!ATTLIST out name CDATA #REQUIRED>
<!ATTLIST out type CDATA #REQUIRED>
<!ATTLIST out sync (sync | async) "sync">
<!ELEMENT instance (portmap*)>
<!ATTLIST instance name CDATA #REQUIRED>
<!ATTLIST instance type CDATA #REQUIRED>
<!ELEMENT portmap (port,%expr;)>
<!ELEMENT op (%expr;,%expr;)>
<!ATTLIST op name
(and | or | xor | plus | minus | mult | div | mod)
#REQUIRED>
<!ELEMENT op1 (%expr;)>
<!ATTLIST op1 name
(not)
#REQUIRED>
<!ELEMENT port EMPTY>
<!ATTLIST port type (self | in | instance) #REQUIRED>
<!ATTLIST port name CDATA #REQUIRED>
<!ATTLIST port instance CDATA #IMPLIED>
<!ELEMENT const EMPTY>
<!ATTLIST const type CDATA #REQUIRED>
<!ATTLIST const value CDATA #REQUIRED>
<!ELEMENT switch (condition, case+ , default?)>
<!ELEMENT condition (%expr;)>
<!ELEMENT case (const,%expr;)>
<!ELEMENT default (%expr;)>
```

Fig. 3. DTD definitions of the XML intermediate representation code

```
1  <out name="oA" type="Logic" sync="sync">
2   <op1 name="not">
3    <op name="and">
4     <op name="or">
5      <port type="in" name="iA" />
6      <port type="in" name="iB" />
7     </op>
8     <port type="in" name="iC" />
9    </op>
10  </op1>
11 </out>
```

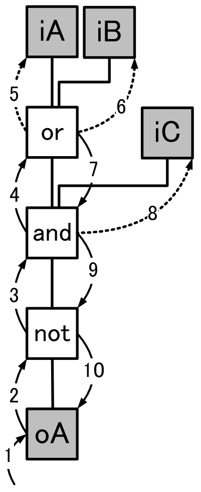


Fig. 4. Example of Node instance connection for output port definition

table. By tracing the circuit configuration management table, a component on the Netlist can be accessed uniquely. The circuit configuration management table has the lists of the following instances: (1) Node instances representing an input port by component definition, (2) Node instances representing an output port, (3) Node instances representing an I/O port of instances in a component definition, and (4) instances expanding in a component.

C. Construction of Circuit Structure

As shown in Figure 3 in the form of source code, in the XML intermediate representation code, all the circuit descriptions are included in the component definitions. The component definitions can be roughly divided into three types; input port definition (`<in />`), output port definition (`<out ></out>`) and instance definition (`<instance></instance>`).

For an input port definition, a port name described in a tag and a component name with its data type and input definition are provided to the Node class to generate an instance. The address information of the generated instance is added to the circuit configuration management table as a definition of the relevant component.

An output port definition contains the connection information for the construction of a logical path to a logic gate extending from the defined output or to the input port. Figure 4 shows an example in which the connection information described in the output port definition is analyzed and expanded. Although the source code in Figure 4 is indented, it is processed for readability and the original XML intermediate representation code is not indented. The connection information has a nested structure where the information appears from the left side in ascending order according to the distance from the output definitions. The information at the far right of the nested structure must be a pair of port definitions.

The processing for the output port definition generates the Node instance representing an output port and constructs a logical path while generating and connecting the Node instances representing the logic gates based on the connection information. First, a Node instance corresponding to the output port definition is generated and its address is added to the circuit configuration management table. For the subsequent connection information, they are processed by a unit defined by two tags and in the order of appearance. For the connection information representing a logic gate, a Node instance is generated as a logic gate and the instance is connected to the path being created. For the connection information representing a port definition, the port represented by the information is traced on the circuit configuration management table and the path is connected to the path being created to terminate the path. When one of the close tags of the connection information representing a logic gate (`</op>` and `</op1>`) appears, go back to the last Node instance before the Node instance of the path being currently created. When the processing proceeds to the close tag of the output port definition (`</out>`), complete the construction of all the paths extending from the output port.

For an instance declaration, the component definition specified by a declaration in the Netlist is referred. The structure in the component definition referred is duplicated at the place of the instance declaration. However, the intermediate code generator of Melasy+ does not hold the order of the definitions of the components in the source code during the code generation. Therefore, when an instance declaration appears in the XML intermediate code, there is a possibility that the component to be instantiated has not defined yet. The Netlist generator expands the definitions of the components, in ascending order of their dependency on the other components, to the XML intermediate representation code using a hierarchical structure analysis function described below. By expanding in ascending order of the independency on the other components, it is possible to avoid a situation where the target component has not defined yet when it is referred.

IV. Static Structure Analysis

A. Check of Circuit Description

As the scale and complexity of a circuit increase, it becomes more difficult for the designer to grasp the whole circuit description. Additionally, since Melasy+ allows the use of control syntax such as ”for and if” statements, it is not possible to check a specific port name and the like at the stage when the Melasy+ code is written. By exploring the Netlist generated, it is possible to check the configuration information of the circuit described. By scanning the circuit configuration management table, it is possible to check the port names for which the problem of label name has been resolved and the unused parts of the circuit description.

B. Analysis of Component Hierarchical Structure

When another component definition is expanded in the component definition by an instance declaration, the component to be expanded cannot reproduce the circuit structure properly if all the elements have been defined. Therefore, the Netlist must be generated with the components in ascending order of their dependency on the other components. It is necessary to analyze the circuit structure to reveal its hierarchical structure of the components.

The component definition depends on the other components when it contains the instance declaration. The instance declaration of the XML intermediate representation code is extracted and the Node instance representing the instance declaration is generated. By adding the Node instance generated to the circuit configuration management table and showing which instance declaration is included in the component definition, the hierarchical structure is grasped. The Node instance representing the instance declaration traces the instance declaration contained in the component definition referring the former declaration on the circuit configuration management table. By connecting to the Node instance traced, it is possible to trace the dependence relationship of the components through the connections of the instance declarations.

C. Count of Logic Gate Stages

A critical path can be derived by counting the stages of the logic gates for each component definition. In each component definition of the Netlist, the order in which the logic is determined by tracing a node instance from an input port is held at each Node instance. When the path encounters a binary operator, the input side with a longer path is selected to determine the value. After the values are set for all the output ports, the longest path is reported as the critical path of the component. Figure 5 shows an example in which the stages are counted for all the adders.

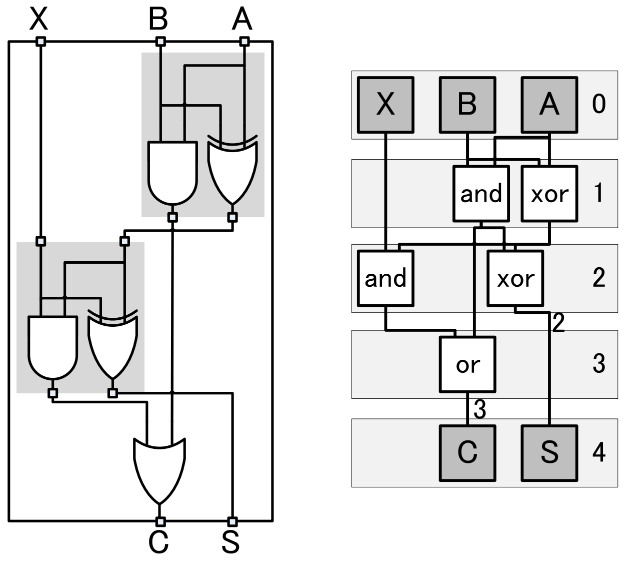


Fig. 5. Count of stages for all the adders

D. Detection of Asynchronous Loop Structure

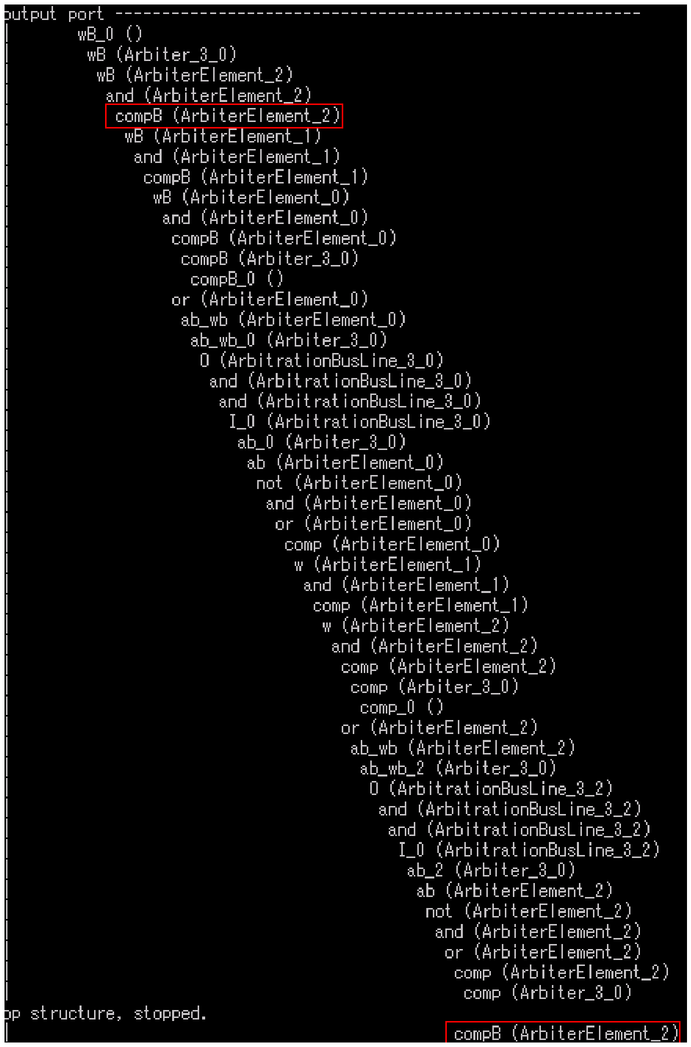
An asynchronous loop structure is a structure which has its value on the circuit structure or has at least one element whose input is determined by its value. When an asynchronous loop structure exists, the operation result is not stabilized and the value continues to change. Asynchronous loop structures in the same component definition can be detected by semantic analysis, but those across multiple component definitions cannot be detected. By carrying out the static structure analysis of the Netlist, it is possible to detect asynchronous loop structures. For a global output port of the Netlist, if the same Node instance is passed twice while scanning all the paths to an input port, there is an asynchronous loop structure which includes the part.

Specifically, Figure 6 shows the result of asynchronous loop structure detection experiment for a high-performance bus arbiter circuit [16] by static structure analysis. Figure 7 shows the image of the detected asynchronous loop structure.

V. Logic Cycle Simulator

A. Purpose and Scope of Logic Simulator

A zero-delay logic cycle simulator function for Netlist was designed as a function to check the behavior of the circuit described. The logic simulation in this study is a zero-delay simulation to track the logic transition of a circuit using a cycle time in a virtual unit time by assuming that all the delay times of the basic logic gates are constant and all the delay times of the connecting signal lines are zero. It is also assumed that there is a sufficient time interval between an input and the next input. Due to these characteristics, a circuit structure which contains an asynchronous loop structure cannot be simulated.



```
utput port --------------------------------------------------------
|       wB_0 ()
|        wB (Arbiter_3_0)
|         wB (ArbiterElement_2)
|          and (ArbiterElement_2)
|           compB (ArbiterElement_2)
|            wB (ArbiterElement_1)
|             and (ArbiterElement_1)
|              compB (ArbiterElement_1)
|               wB (ArbiterElement_0)
|                and (ArbiterElement_0)
|                 compB (ArbiterElement_0)
|                  compB (Arbiter_3_0)
|                   compB_0 ()
|                 or (ArbiterElement_0)
|                  ab_wb (ArbiterElement_0)
|                   ab_wb_0 (Arbiter_3_0)
|                    0 (ArbitrationBusLine_3_0)
|                     and (ArbitrationBusLine_3_0)
|                      and (ArbitrationBusLine_3_0)
|                       I_0 (ArbitrationBusLine_3_0)
|                        ab_0 (Arbiter_3_0)
|                         ab (ArbiterElement_0)
|                          not (ArbiterElement_0)
|                           and (ArbiterElement_0)
|                            or (ArbiterElement_0)
|                             comp (ArbiterElement_0)
|                              w (ArbiterElement_1)
|                               and (ArbiterElement_1)
|                                comp (ArbiterElement_1)
|                                 w (ArbiterElement_2)
|                                  and (ArbiterElement_2)
|                                   comp (ArbiterElement_2)
|                                    comp (Arbiter_3_0)
|                                     comp_0 ()
|                                   or (ArbiterElement_2)
|                                    ab_wb (ArbiterElement_2)
|                                     ab_wb_2 (Arbiter_3_0)
|                                      0 (ArbitrationBusLine_3_2)
|                                       and (ArbitrationBusLine_3_2)
|                                        and (ArbitrationBusLine_3_2)
|                                         I_0 (ArbitrationBusLine_3_2)
|                                          ab_2 (Arbiter_3_0)
|                                           ab (ArbiterElement_2)
|                                            not (ArbiterElement_2)
|                                             and (ArbiterElement_2)
|                                              or (ArbiterElement_2)
|                                               comp (ArbiterElement_2)
|                                                comp (Arbiter_3_0)
|
p structure, stopped.
|                                              compB (ArbiterElement_2)
```

Fig. 6. Example of detection of asynchronous loop structure

Furthermore, a gate stage number table is prepared that holds the order in which the logic gates determine the logic and manages the addresses of all the elements and the number of the steps using the count function of logic gate stages described above. The values of the inputs and outputs to be handled are true, false and unknown.

B. Change of Output to Input

Observation of the logic transition is carried out by providing a virtual unit time simulating the clock and an input scenario to the Netlist. Along with the virtual unit time based on the input scenario, an input is provided to a globally accessible input port.

The procedure of the logic simulation is as follows. First, when the virtual unit time is 0, check if an input scenario specifying the initialization exists. When there is an input scenario specifying initialization, change the output of the port specified according to the input scenario. According to the input of the initialization, all the elements carry out their operation in the order in which the logic is determined based on the gate stage number table. If the initialization is not carried out, the output of the port is unknown at the initial state. All the elements access to the elements from which they receive their inputs, in the order in which the logic is determined, and check if there is an output. If there is an output, the value being output is taken as its input. When an element has an input and the operation is carried out, the result is output to substitute it to its output value.

C. Description of Input Scenario

Observation of the logic transition is carried out by providing a virtual unit time simulating the clock and an input scenario to the Netlist. Along with the virtual unit time based on the input scenario, an input is provided to a globally accessible input port.

The virtual unit time is given by a natural number starting from 0. It is only used for the purpose to manage the order in which the inputs are provided. It is assumed that there is a sufficient time interval from an input to the next input and all the operations are completed when the next input is provided. There is no restriction on the input scenario that it must be described in the order of virtual unit time.

The input port name is given by the port name used by the XML intermediate code. The label must be one for which the problem has been resolved. The input port described in the input scenario is limited to one that can be accessible globally.

The value must be provided either true or false. It is not assumed in the input scenario.

VI. Conclusions and Future Work

A Netlist which has a specific structure of circuit description was generated from the Melasy+ intermediate code. By carrying out the static structure analysis of the Netlist generated, it was verified that the check of circuit description, analysis of component dependence relationship, count of gate stages, and detection of asynchronous loop structure are possible. To check the behavior of circuit description, a logic simulator using the Netlist was designed.

By enabling the check and static structure simulation of circuit description with Melasy+ only, it is possible to reflect the analysis result to upper-level design and improve design at an earlier stage.

First, the future challenge is to evaluate the series of design and verification tools, which are the deliverables of this study, using actual size problems. Evaluation through the comparison of the design efficiency using the existing hardware design environment and that using the environment provided this study, as well as evaluation of the change in calculation time of the static structure analysis function due to increased circuit scale, are planned.

Next, we plan to expand the static structure analysis function. A logic simulation function will be implemented and the False-path detection function and the optimum solution presentation function for cutting of asynchronous loop structure will be expanded. The abstract circuit information obtained

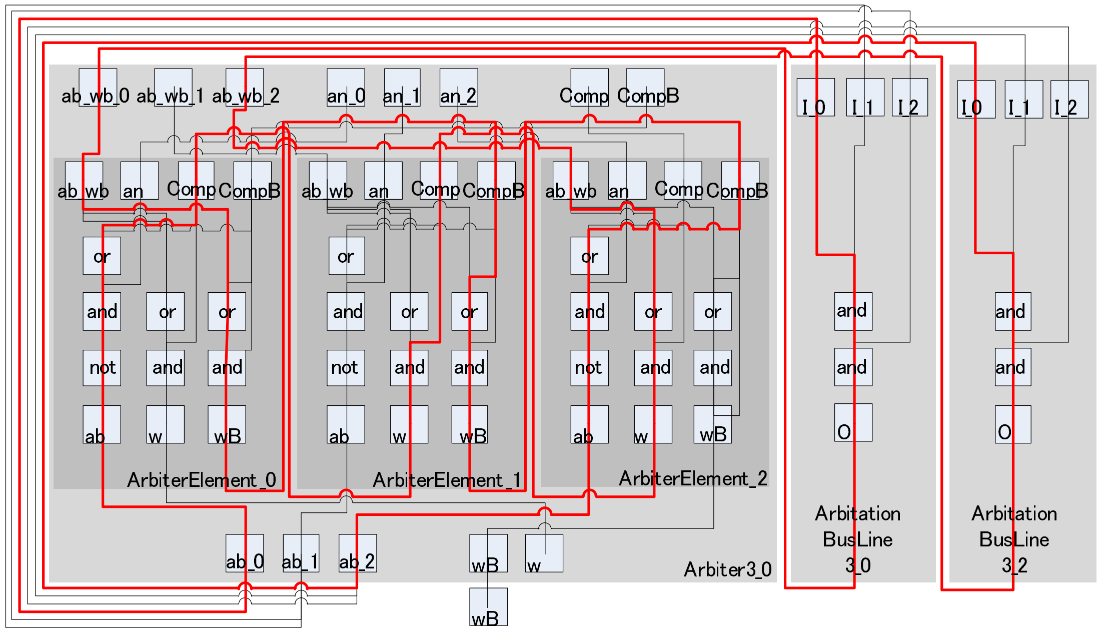


Fig. 7. Example of detection of asynchronous loop structure for Netlist

from the analysis function will be held by component definition and the circuit information with an appropriate level of abstraction will be provided when needed. Furthermore, we also plan to expand the Netlist. Finally, we will redesign the Melasy+ compiler. Essentially, a circuit representation like a Netlist can be obtained when an intermediate representation code is generated. We aim to design an intermediate representation that holds much information than that held by an XML text format and implement a better code generator.

ACKNOWLEDGEMENT

This research was partially supported by the Ministry of Education, Science, Sports and Culture of Japan, Grant-in-Aid for Scientific Research (C), 23500174, 2011.

REFERENCES

[1] Y. N. Patt, S. J. Patel, M. Evers, D. H. Friendly, and J. Stark, “One billion transistors, one uniprocessor, one chip,” IEEE Computer, 30(9):5157, 1997.
[2] N. Wirth, “Digital Circuit Design,” Springer, New York, NY, USA, 1985.
[3] A. V. Aho, R. Sethi, and J. D. Ullman, “Compilers: Principles, Techniques, and Tools,” Addison Wesley, Boston, MA, USA, 1986.
[4] A. V. Aho and J. D. Ullman, “Principles of Compiler Design,” Addison Wesley, Boston, MA, USA, 1977.
[5] T. Grotker, “System Design with System-C,” Kluwer Academic Publishers, Norwell, MA, USA, 2002.
[6] H. Trickey, “Flamel: A high-level hardware compiler,” IEEE Trans. on Comp.-Aided Design of Int. Circ. and Sys.,6(2):259269, 1987.
[7] *HDCaml* : http://www.confluent.org/wiki/doku.php/hdcaml.
[8] *Confluence* : http://www.confluent.org/wiki/doku.php/confluence.
[9] B.Berard, et, al., “Systems and Software Verification Model-Checking Techniques and tools,” Springer, 2001.
[10] E. M. Clarke, O. Grumberg, and D. Peled, “Model Checking,” MIT Press, Boston, MA, USA, 2000.
[11] K. L. McMillan, “The SMV system (Symbolic Model Verifier),” Cadence Berkeley Labs, Berkeley, CA, USA, 1999.
[12] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri, “Nusmv: A new symbolic model verifier,”; In Proc. of 11th Conference on Computer-Aided Verification (CAV’99), pages 495499, 1999.
[13] “VHDL (VHSIC Hardware Description Language),” IEEE Design Automation Standards Committee, 1076, 2002.
[14] D. E. Thomas and P. R. Moorby, “The Verilog(r) Hardware Description Language,” Kluwer Academic Publishers, Norwell, MA, USA, 2002.
[15] N. Iwasaki and K. Wasaki, “A Meta Hardware Description Language *Melasy* for Model Checking Systems,” Proc. of the 5th Int’l Conf. on Information Technology, New Generations(ITNG2008), 273-278, 2008.
[16] K. Tokito, T. Matsubara, and Y. Koga, “Dependable Bus Arbitration by Alternating Competition with Checkers,” IEICE Trans Inf.&Syst., E80-D(1), 44-50, 1997.