Synthesis of Four-Phase Asynchronous Control Circuits from Pipeline Dependency Graphs

Hiroto Kagotani* Takuji Okamoto* Takashi Nanya†

*Department of Communication Network Engineering
Okayama University
Okayama 700-8530, Japan
Tel: +81-86-251-{8184,8181}
Fax: +81-86-251-8255
e-mail: {kagotani,okamoto}@cne.okayama-u.ac.jp

†Research Center for Advanced Science and Technology
The University of Tokyo
Tokyo 153-8904, Japan
Tel: +81-3-5452-5160
Fax: +81-3-5452-5161
e-mail: nanya@hal.rcast.u-tokyo.ac.jp

Abstract — We propose a method of synthesizing pipeline controllers as four-phase asynchronous circuits from specifications described as two-phase dependency graphs. Pipeline two-phase dependency graphs are transformed into four-phase ones by applying a transformation rule to each simple loop in the graphs. Four-phase dependency graphs are easily mapped onto four-phase asynchronous control circuits. We also discuss some simplification of four-phase dependency graphs.

I. Introduction

Asynchronous circuits are expected to be an effective approach to high-speed and/or low-power processors. There have been a lot of works done for asynchronous design [1] including synthesis methods of controllers [2, 3, 4], and pipeline controller design [5, 6, 7, 8].

We have proposed a synthesis method of four-phase asynchronous controllers that controls four-phase resources based on mapping of dependency graphs, which represent the execution orders of micro-operations[9, 10][1]. Execution of a micro-operation implemented by a four-phase resource is decomposed into cyclic execution of a working phase, a stable phase, an idle (i.e. return-to-zero) phase and another stable (i.e. spacer) phase.

Although dependency graphs are capable of describing pipeline execution of micro-operations, the mapping is applicable only to non-pipeline dependency graphs since, to implement the specified execution order, the mapping takes advantage of the fact that whole idle phases can be safely executed after the completion of whole working phases in non-pipeline circuits.

STGs can describe such four-phase pipeline, but writing STGs describing such complex behavior is not considered to be acceptably easy. Adding two-phase (transition signaling logic) to four-phase (level sensitive logic) converters to each micro-operation may solve the problem since two-phase dependency graphs can be easily mapped onto two-phase control circuits. This approach, however, may need larger hardware because such a converter have to implement a finite state machine.

In this paper, we propose a synthesis method of four-phase asynchronous controllers from dependency graphs describing pipeline execution. We assume that micro-operations that can be executed in parallel in the given dependency graphs use separate functional resources so that we can ignore the arbitration problem. In Section II, we clarify the problem concerned with mapping of dependency graphs. In Section III, we propose the intermediate form of graph, four-phase dependency graph, and describe the transformation from original to four-phase dependency graphs.

[1] Although we used to call asynchronous circuits using return-to-zero handshakes two-phase or four-cycling, we call them four-phase in this paper.

II. Two-phase Dependency Graphs

A. Circuit synthesis using dependency graphs

We target synthesis of four-phase asynchronous (speed-independent or quasi-delay-insensitive[11]) control circuits. They control four-phase asynchronous data-paths (resources) in the four-phase request-acknowledgment manner.

Without loss of generality, an asynchronous data-path used for a micro-operation (register-to-register data transfer) can be modeled as follows. The request signal triggers the start of the operation, and the acknowledgment signal represents its completion. When the request signal is reset, the acknowledgment signal is also reset without affecting the corresponding data-path. We call the former period a working phase and the latter period an idle phase.

We have proposed a model called dependency graph in [9, 10] as a variant of control/data-flow graphs to describe the specification of asynchronous control circuits. It is also regarded as a subclass of Petri nets or free-choice nets. Although it is capable of describing conditional branches and loops, we exclude the capability in this paper to simplify the problem. Fig. 1(a) shows an example of a dependency graph.

A dependency graph is a set of four types of nodes and directed edges between them. A node represents an operation. A directed edge represents the execution order of the operations denoted by its start and end nodes. Execution of an operation

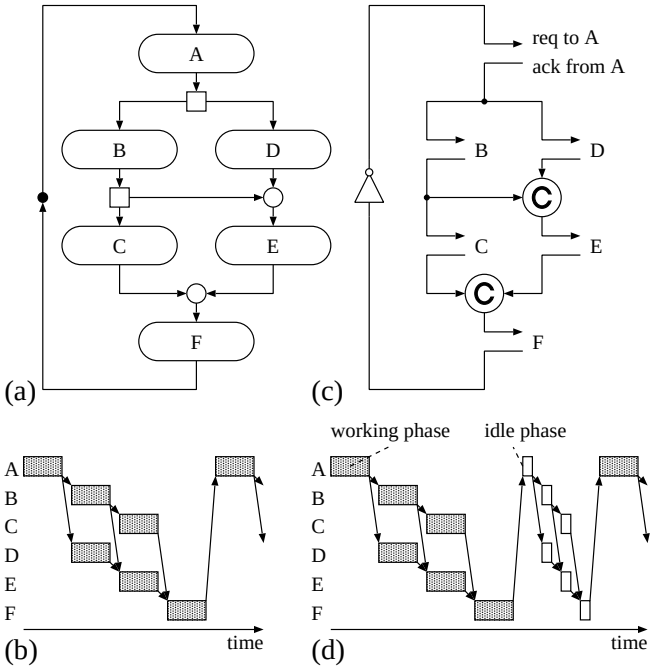

Fig. 1. Example of a dependency graph.

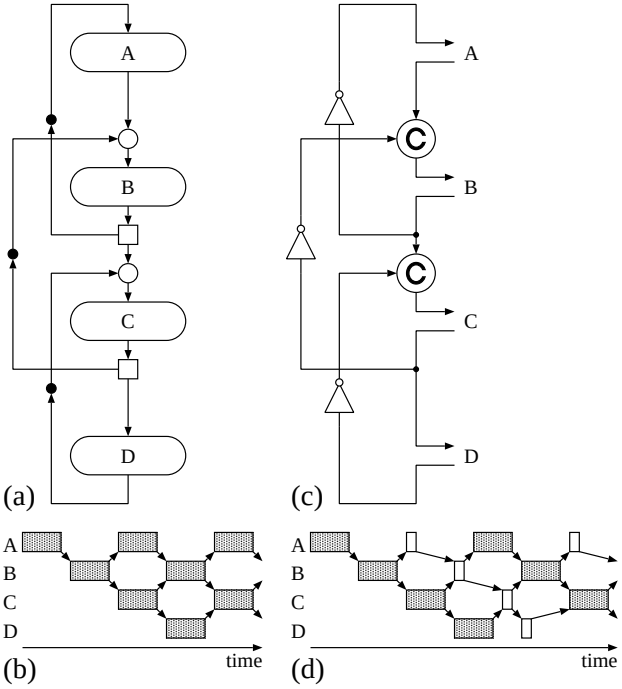

Fig. 2. Example of a pipeline dependency graph.

is controlled by tokens moving along edges.

Each type of nodes operates as follows. An oval node denotes a micro-operation. It starts execution when a token arrives in its input edge and transfers the token to its output edge when the execution completes. A small square (fork) node and a small circle (join) node denote the start and the end of parallel processing respectively. When a token arrives in the input edge of a fork node, it removes the token and puts a token on every output edges. Conversely, when tokens arrive in all input edges of a join node, it removes them and put a token on the output edge. A black dot denotes an initial position of a token. It sends a token at the beginning to its output edge, and then transfers every input token to the output. A graph must be so constructed as to satisfy the condition that any edge does not hold two or more tokens at any time.

Fig. 1(b) shows an example of execution timing of micro-operations in (a). A box denotes that the corresponding operation is executed in the period. Arrows denote causality between operations. B and D can be executed in parallel, and so can C and E. C can start when B completes, and E can start when both B and D complete.

The graph shown in Fig. 1(a) can be mapped onto a circuit shown in (c). Edges, fork nodes, join nodes and token initial positions correspond to wires, wire branches, Muller's C elements and inverters respectively. Micro-operations correspond to request-acknowledgment interconnections with the data-paths.

Although tokens represent only the execution of working phases of micro-operations, idle phases of all micro-operations can be interpreted to be executed when tokens return to their initial positions. The execution timings of the graph and the circuit (Fig. 1(b) and (d) respectively) are obviously equivalent except for the execution of idle phases in (d).

B. Mapping problem of pipeline dependency graphs

Dependency graphs can model not only simple parallel execution but also pipeline execution. Fig. 2(a) and (b) show an example of a pipeline dependency graph and its execution timing respectively. Micro-operations {A, C} and {B, D} are alternately executed in parallel except for some transient period.

If we apply the mapping method described in Section II.A to the pipeline dependency graph, we can obtain the mapped circuit shown in Fig. 2(c) and its execution timing in (d). Obviously, timing in (b) and (d) are not equivalent. For example, B and D are executed in parallel in (b), but their working phases can not in (d). Instead, an idle phase of B and a working phase of D are executed in parallel. This means that the mapping is not applicable to pipeline dependency graphs.

The reason of the inequality is that two-phase (conventional) dependency graphs do not distinguish two types of events, namely, 0-to-1 and 1-to-0 transitions on wires in mapped circuits. For pipeline control circuits, idle phases can no longer be interpreted to be executed implicitly, since other tokens affect their execution timing. For example, even if the token in the first loop returns to its initial position, an idle phase of B can not start execution because it is also controlled by the token in the second loop. For non-pipeline control circuits, there is no such interaction of loops in dependency graphs.

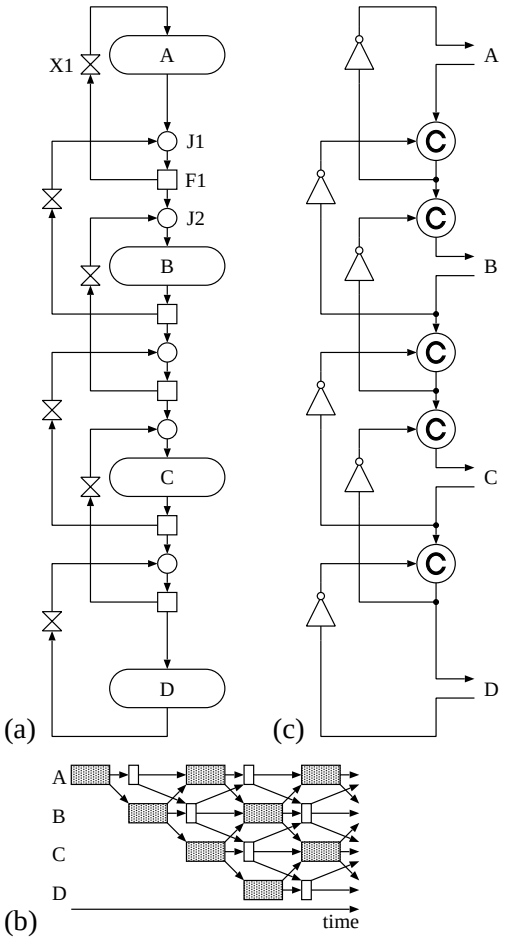

Fig. 3. Example of a pipeline four-phase dependency graph.

III. Graph Transformation from Two-phase to Four-phase

A. Four-phase interpretation of dependency graphs

A straightforward solution to the mapping problem mentioned above is to change interpretation of dependency graphs so that they represent four-phase execution. We call such a graph with four-phase interpretation a four-phase dependency graph. Fig. 3(a) shows an example of a pipeline four-phase dependency graph.

A four-phase dependency graph consists of almost the same types of elements as two-phase one. Only for distinction between their appearances, we use ⋈ symbols instead of black dots. Working phases and idle phases of an operation are controlled by two types of tokens, namely, working tokens and idle tokens respectively. A ⋈ symbol denotes not only an initial position of tokens, but a token exchanger. It sends a working token at the beginning to its output edge, and then inverts every input token to the opposite type while transferring it to the output. A four-phase dependency graph must be so constructed as to satisfy the conditions that any edge does not hold two or more tokens at any time, and input edges of any join node do not hold different types of tokens at any time.

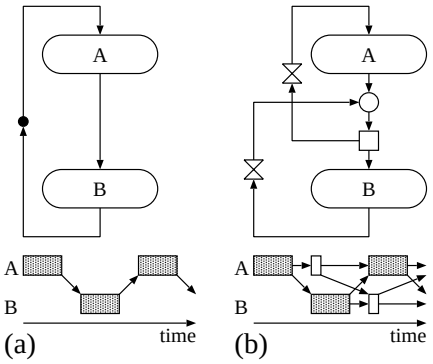

Fig. 4. Transformation of a single loop graph.

Fig. 3(b) shows the execution timing of the graph in (a). For example, at first, only the first working phase of A is executed, though every token exchanger outputs a working token. After that, the token arrived in the output edge of A is transferred through join node J1, fork node F1 and join node J2, and starts a working phase of B. On the other hand, the working token copied at F1 is inverted to an idle token at X1 and the idle token starts an idle phase of A. As the result, the first idle phase of A and the first working phase of B are executed in parallel.

A four-phase dependency graph models a level sensitive asynchronous control circuit more exactly. Therefore, the circuit mapped from the graph using the same mapping method as that for two-phase graphs operates on the execution timing equivalent to the original graph even if it describes pipeline operations. For example, Fig. 3(c), which shows the mapped circuit from (a), operates on the timing in (b).

B. Transformation rule

It is, however, not a trivial problem to compose a four-phase dependency graph that operates as intended. We consider obtaining four-phase dependency graphs by transforming two-phase ones.

Consider the two-phase dependency graph shown in Fig. 2(a). It intends A and C to be executed in parallel, i.e., working phases of A and C must be executed in parallel in the transformed four-phase dependency graph. It is natural that an idle phase of B is executed in parallel with working phases of A and C, though its execution timing has some choice. Hence we intend to compose a four-phase dependency graph so that the opposite phases of A and B, B and C, and C and D are executed in parallel.

Consider a simpler two-phase dependency graph in Fig. 4(a). It consists of a single loop with micro-operations A and B. Fig. 4(b) shows a four-phase dependency graph so composed as to execute the opposite phases of A and B in parallel. Since the graph in Fig. 2(a) consists of simple loops including one token initial position, an equivalent four-phase dependency graph can be constructed by transforming every loop into the form of Fig. 4(b) and composing them again by unifying nodes

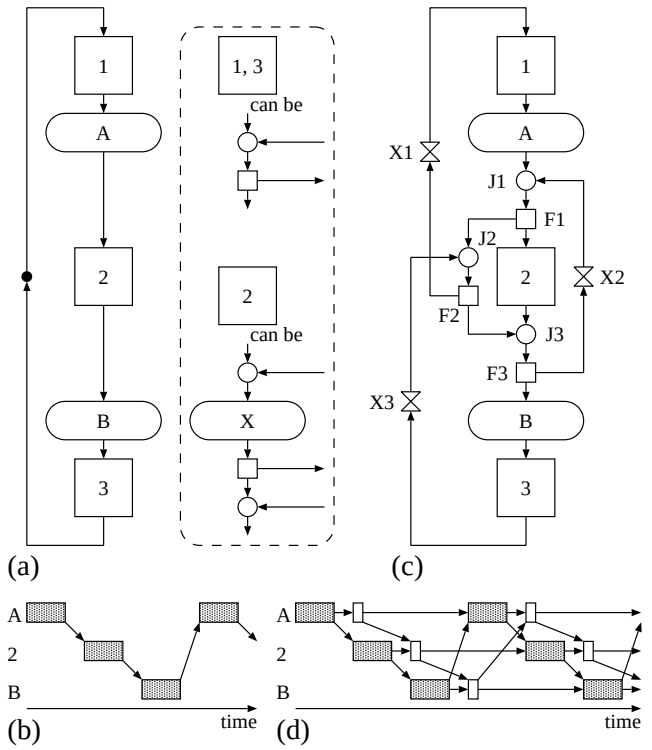

Fig. 5. Transformation of a generic simple loop.

with the same label. The graph in Fig. 3(a) can be constructed from Fig. 2(a) in this way.

In general, each simple loop in a two-phase dependency graph may contain any number of micro-operation, fork and join nodes. Without loss of generality, we can describe such a simple loop as shown in Fig. 5(a). Node A and B represent the first and the last micro-operations respectively that a token passes through while it travels round the loop. Boxes 1, 2 and 3 represent sub-paths, i.e., parts of the sequence of edges and nodes in the loop. According to the definition of A and B, boxes 1 and 3 does not include any micro-operation, while box 2 may include micro-operations. The area surrounded with dashed line shows examples of the contents of such boxes. Fork and join nodes in the boxes are connected with other nodes in any boxes, or in other parts of the graph. If a loop include less than two micro-operation, we insert one or two dummy micro-operations next to the token initial position so that the loop fits the form of the figure. We provide this rule simply for generality, but such a loop is redundant and should be eliminated ahead of time unless it is the only loop in the graph. Fig. 5(b) shows the execution timing of A, B and micro-operations in box 2. The number of micro-operations in box 2 affects only the length of the boxes located in the row 2.

Fig. 5(c) shows the proposed four-phase dependency graph corresponding to the generic simple loop in (a). It is generated by adding a bypass from F1 to J3, adding two feedback paths like Fig. 4 using the bypass, and adding the third feedback path from F3 to J1 via X2. The last feedback path is required so that the graph satisfies the condition mentioned in Section III.A, i.e., if it did not exist, a token copied in F1 might pass through X1 and A and catch up with another token in box 2. Its execution timing is shown in Fig. 5(d). The timing of working phases is equivalent to (b). Idle phases can be executed in parallel with as many micro-operations as possible. Any idle phase, however, is not executed in parallel with any other idle phase since working and idle phases represented by four-phase dependency graphs are symmetric.

Applying this transformation to a simple loop in a two-phase dependency graph does not affect the connectivity of this loop and other parts of the graph, since the transformation does not change the contents of boxes, which are the only parts in the loop connected to other parts. We can therefore safely apply the transformation to every simple loop without affecting any connectivity. Hence, four-phase dependency graphs can be obtained by applying the transformation to every simple loop in the given two-phase dependency graphs.

The outline of the transformation algorithm can be described as follows:

1. Finding every simple loop in the specified dependency graph.
2. For each simple loop:
   (a) Matching the loop to the pattern of Fig. 5(a).
   (b) Transforming the loop to the form of Fig. 5(c).

Obviously, in the way of the transformation, the graph being transformed includes two-phase part and four-phase part at the same time. That does not, however, prevent the algorithm from being applied because any four-phase part in a two-phase simple loop is hidden in the boxes 1, 2 or 3.

Although we can prove that the execution timing of the transformed four-phase dependency graph is equivalent to that of the two-phase one, space did not permit us to describe the proof.

C. Simplification of four-phase dependency graphs

The transformation described above adds three fork nodes, three join nodes and three token exchanger while removing one token initial position for each simple loop. Transformed graphs may include some redundancy, which should be eliminated for reducing hardware volume of the mapped circuits.

If box 2 is empty, the path from F1 to J3 is redundant and can be eliminated, and therefore the feedback path from F3 to J1, which is for avoidance of catching up in box 2 mentioned above, can also be eliminated. Further, if box 2 consists of a sequence of any number of fork nodes followed by a sequence of any number of join nodes, the fork nodes and the join nodes can be unified with F1 and J3 in Fig. 5(c) respectively, and the path from F1 to J3 and the feedback path from F3 to J1 can be eliminated because of the same reason.

Applying the transformation and simplification rules, we can obtain the four-phase dependency graph in Fig. 3(a) from the two-phase one in Fig. 2(a).

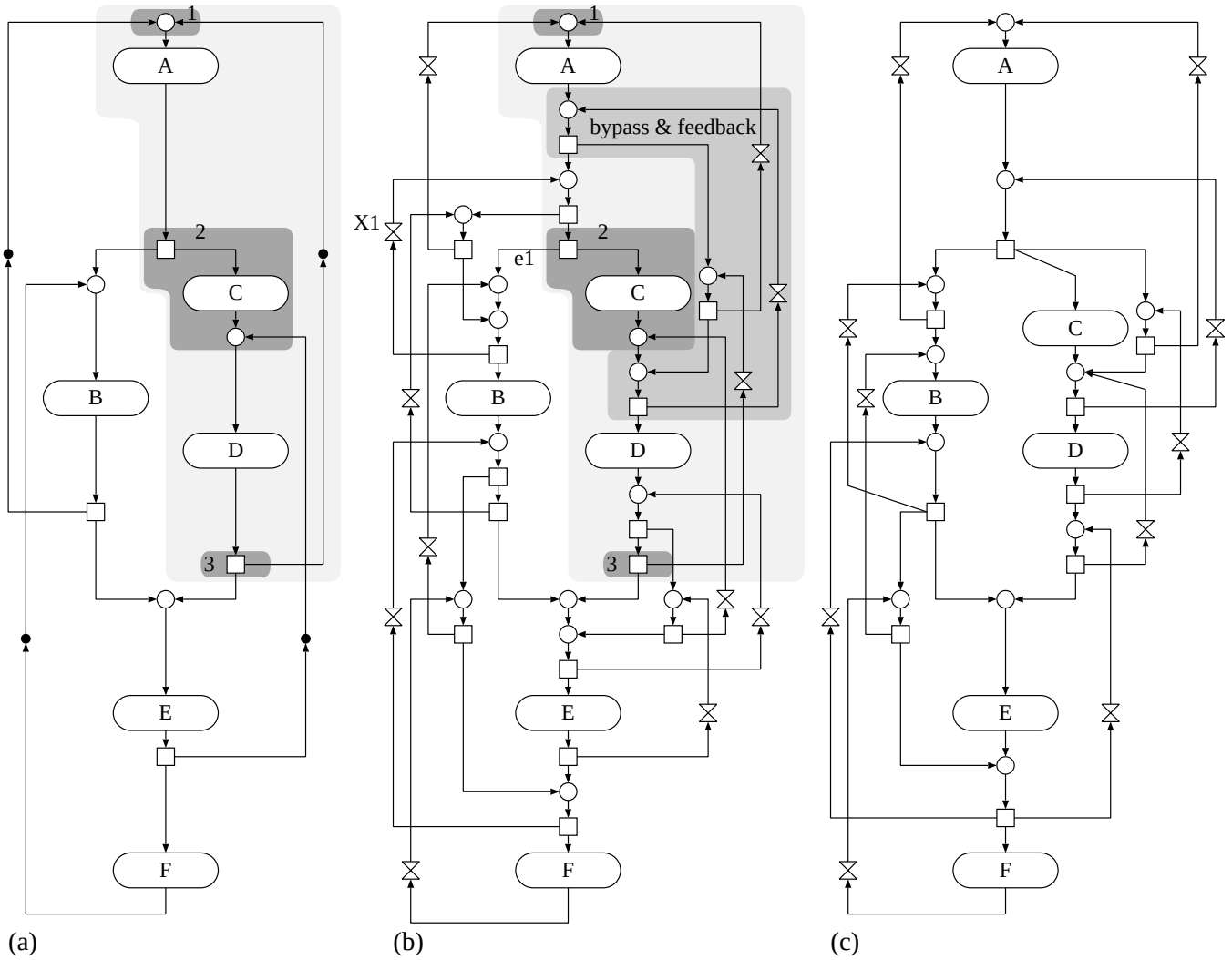

Fig. 6. Example of transformation from two- to four-phase dependency graph.

D. Example

Consider the pipeline two-phase dependency graph shown in Fig. 6(a). It specifies not only parallel execution of B and (C, D), but pipeline execution of A and (E, F), (A, C) and E, and D and F.

We first transform the simple loop indicated as the light gray region. The dark gray regions correspond to the boxes 1, 2 and 3 in Fig. 5(a). By applying the transformation rule described above, the loop is transformed into the light gray region of Fig. 5(b). The medium gray region represents nodes and edges added by the transformation. Other fork and join nodes in the light gray region are added by transformation of other loops. Note that the transformation does not change the dark gray regions. By applying the transformation rule to three other simple loops, we obtain Fig. 6(b) as the four-phase dependency graph.

The graph includes some redundancy. For example, since edge e1 is redundant, the edge and the feedback path including X1 can be eliminated. The four-phase dependency graph shown in Fig. 6(c) is obtained by applying such simplification. This graph can be easily mapped onto a four-phase asynchronous control circuit using the correspondence described above.

IV. Conclusion

We have proposed a method of synthesizing pipeline controllers as four-phase asynchronous circuits from specifications described as two-phase (conventional) dependency graphs. Pipeline two-phase dependency graphs are transformed into four-phase ones by applying a transformation rule to each simple loop in the graphs. four-phase dependency graphs are easily mapped onto four-phase asynchronous control circuits. We have also discussed some simplification of four-phase dependency graphs.

We have not discussed transformation of dependency graphs that include conditional branches. To synthesize more practical circuits, we have to expand the method to such classes

of dependency graphs. It may be achieved by modifying the transformation rule so that it can handle more generic form of loops including branching nodes.

This research was supported in part by the Ministry of Education of Japan under scientific-research grant-in-aid 09780288 and 11780225.

REFERENCES

[1] Erik Brunvand, Steven Nowick, and Kenneth Yun. Practical advances in asynchronous design and in asynchronous/synchronous interfaces. In *Proc. ACM/IEEE Des. Automation Conf.*, pages 104–109, 1999.

[2] Tam-Anh Chu. Synthesis of self-timed VLSI circuits from graph-theoretic specifications. In *Proc. Int. Conf. Comput. Des. (ICCD)*, pages 220–223. IEEE Comput. Society Press, 1987.

[3] Teresa H.-Y. Meng, Robert W. Brodersen, and David G. Messerschmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Trans. Comput.-Aided Des.*, 8(11):1185–1205, November 1989.

[4] Kees van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[5] Ivan E. Sutherland. Micropipelines. *Commun. ACM*, 32(6):720–738, June 1989.

[6] K. Y. Yun, P. A. Beerel, and J. Arceo. High-performance asynchronous pipeline circuits. In *Proc. Int. Symp. Advanced Research in Asynchronous Circuits & Syst.* IEEE Comput. Society Press, March 1996.

[7] D. A. Gilbert and J. D. Garside. A result forwarding mechanism for asynchronous pipelined systems. In *Proc. Int. Symp. Advanced Research in Asynchronous Circuits & Syst.*, pages 2–11. IEEE Comput. Society Press, April 1997.

[8] K. Y. Yun, P. A. Beerel, and J. Arceo. High-performance two-phase micropipeline building blocks: double edge-triggered latches and burst-mode select and toggle circuits. *IEE Proc., Circuits, Devices & Syst.*, 143(5):282–288, October 1996.

[9] Takashi Nanya, Yoichiro Ueno, Hiroto Kagotani, Masashi Kuwako, and Akihiro Takamura. TITAC: Design of a quasi-delay-insensitive microprocessor. *IEEE Des. & Test Comput.*, 11(2):50–63, 1994.

[10] Hiroto Kagotani and Takashi Nanya. A synthesis method of quasi-delay-insensitive processors based on dependency graph. In *Asia-Pacific Conf. Hardware Description Languages (APCHDL)*, pages 177–184, October 1994.

[11] Scott Hauck. Asynchronous design methodologies: An overview. *Proc. IEEE*, 83(1):69–93, January 1995.