2H－3

A Method for Generating Asynchronous Pipeline Circuits from Dependency Graph

Rafael K. Morizawa Yoichiro Ueno Takashi Nanya†

Tokyo Institute of Technology, Graduate School of Information Science and Engineering
†University of Tokyo, Research Center for Advanced Science and Technology

1 Introduction

Pipelines are well known structures that increase a VLSI system's speed and throughput. In this note we propose a method for generating two-rail four-cycling handshaking asynchronous pipeline circuits from Dependency graphs. The method generates an application-specific hardware where control flow and data flow elements coexist.

2 Preliminaries

The Dependency graph considered here is a directed graph with 5 types of nodes (namely *micro-operation node*, *fork node*, *join node*, *select node*, and *merge node*), arcs, and tokens [1]. A Dependency graph that specifies a division algorithm is shown in figure 1. Nodes represent an operation or control flow. The arc between two nodes represents the dependency relation between the two. The Dependency graph has only one input arc (*primary input*) and only one output arc (*primary output*).

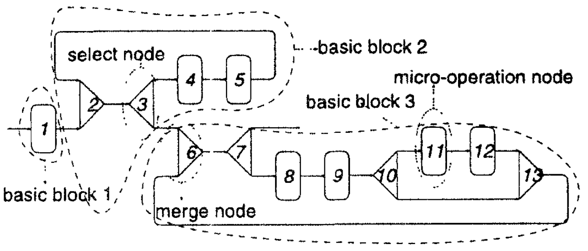

Figure 1: An example Dependency graph.

We define, for the Dependency graphs D that the algorithm deals with, the concept of *basic block*. It is a subgraph B that can be obtained by the application of the following procedure. Let $b_h \in B$ and $b_t \in B$ such that b_h is the **only** node in B to where arcs from other subgraphs $B' \neq B$ enter (the *head node* of B), and b_t is the **only** node in B from where arcs to other subgraphs $B'' \neq B$ leave (the *tail node* of B). Let b be the node where the primary input arc is one of its input arcs.

1. Starting from b, find all the maximal strongly connected subgraphs of the Dependency graph. Each such subgraph is a basic block. Remove them from D.
2. If there are nodes remaining in D do for each disjoint subgraph D_d:
   (a) Let b_{head} be the node of the subgraph D_d such that the input arcs does not come from any node in D_d.
   (b) Follow the path from the node b_{head}.
      i. If the path has a node b_f where arcs fork (such as a fork or a select node), then the subgraph formed by the path b_{head}–b_f is a basic block. Remove the basic block from D. For each path of b_f let the first node be b_{head} and do step 2b.
      ii. If the path has a node b_j where arcs join (such as a merge or join node), then the subgraph formed by the path b_{head} to the node prior to b_j is a basic block. Remove the basic block from D. Let the b_j be b_{head} and do step 2b.
      iii. If the path finishes in a primary output arc, then the subgraph formed by the path from b_{head} to the current node is a basic block. Remove the basic block from D and repeat step 2b.
   (c) Repeat step 2.
3. End procedure.

The above definition and procedure are based on [2]. In figure 1 the subgraph formed by node $\{1\}$ is a basic block, as well as the subgraph formed by nodes $\{2,3,4,5\}$, and by nodes $\{6,7,8,9,10,11,12,13\}$.

The algorithm presented in the next section generates asynchronous pipeline circuits based on the pipeline structure of the asynchronous RISC microprocessor TITAC-2 [3]. A schematic design of a basic pipeline stage is shown in figure 2. A synthesized pipeline stage schematics can vary depending on the type of Dependency graph nodes from which the pipeline stage was derived.

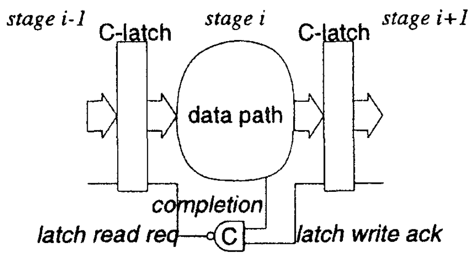

Figure 2: A basic pipeline stage.

3 The algorithm

Provided a Dependency graph with the characteristics described in the previous section, the following algorithm generates an asynchronous pipeline from it. Let B be a basic block, $b_h \in B$ be the head node of B, and $b_t \in B$ be the tail node of B.

1. Divide the Dependency graph into basic blocks. Each basic block corresponds to a pipeline stage.
2. Once the pipeline stages have been defined, placement of the stage latches takes place. The number of latches between basic blocks i and j is equal to the number of variables that are live and cross the boundary between i and j.
3. The pipeline stages are synthesized according to the following set of rules:
   (a) If b_t is a select node and B is strongly connected, then synthesize the pipeline stage using the framework shown in figure 4.
   (b) If b_t is a select node and B is not strongly connected, then synthesize the pipeline stage using the framework shown in figure 3.

(c) If b_h is a merge node, then synthesize using the framework of figure 2, and place a multiplexer between the input stage latches and the data-path, connecting appropriately the control lines.

(d) If b_h is a join node, then synthesize using the framework of figure 2, and place a latch between the input stage latches and the data-path that opens only when there is valid data in all the input stage latches.

(e) Otherwise synthesize the pipeline stage using the framework shown in figure 2.

4. The basic blocks defined as pipeline stages are synthesized, except for those that corresponds to branches or loops.
5. The final step is the placement of the control circuitry in each stage.

In this algorithm loops are synthesized so that the whole loop body fits into one pipeline stage, and the different instances of the loop execution are not overlapped. The framework for loops (figure 4) considers that there is a conditional branch near the head of the basic block. Any loop can be reduced to this structure by making transformations to the subgraph.

The data path is synthesized by using a method similar to [1], where a pre-defined library of functional blocks is used to synthesize data-paths. Interconnection of control lines between the pipeline stage latches and the data path functional blocks is done with the controllers associated with the basic pipeline structures shown in figures 2, 3 and 4, instead of the controllers used in [1].

The conditional branch decision circuits shown in figures 3 and 4 implement the conditional branch of select nodes. They are synthesized using the same approach for synthesizing such circuits in [1]. The output of this circuit is a 1-out-of-n control signal (where n is the number of branches) that controls which pipeline out of the n will be activated.

A simplified diagram of the circuit obtained by applying the above algorithm to the Dependency graph in figure 1 is shown in figure 5. For the sake of simplicity, control structures have been omitted. Arrows represent the flow of data, small rectangles represent a pipeline latch, and the structures labeled "branch" represent the logic that implements branching decision. Ovals represent the synthesized data path in each pipeline stage. Stage 2 corresponds to basic block 2 in figure 1, and stage 3 corresponds to basic block 3. Since in figure 1 basic block 1 only initializes variables, stage 1 consists only of pipeline stage latches with data being input.

4 Discussion

The algorithm defined in the previous section provides a simple methodology to synthesize asynchronous pipelines. For this simplicity, the algorithm can be easily implemented as a tool. A characteristic of the circuits generated by this algorithm is that they are a collection of small pipelines where the output end of one such pipeline is connected to the input end of one or more other pipelines.

5 Conclusion

We have defined a simple, yet easy to implement algorithm for generating asynchronous pipeline circuits from a Dependency graph. So far, the method can deal with a limited set of Dependency graphs.

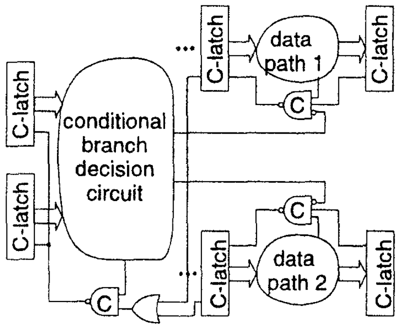

Figure 3: Interface for branching.

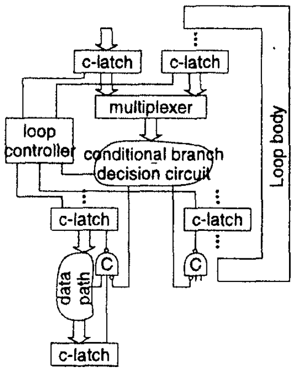

Figure 4: Loop interface.

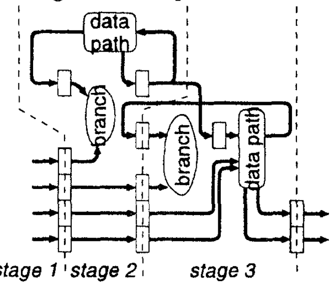

Figure 5: Schematic description of the circuit generated from the Dependency graph in figure 1.

The method itself needs improvements in the following points: (1) an asynchronous loop pipelining algorithm must be added to the main algorithm; (2) determine the exact conditions for the "optimal" throughput pipeline structure. Future work will concentrate on these two points and, in addition, on the implementation of the algorithm and its integration to the tool-set under development.

This work was supported in part by the Ministry of ESSC under Grant-in-aid for Scientific research No. 07558036 and by NEDO under the Proposal-based Advanced Industrial Technology R&D Program.

References

[1] 籠谷裕人, 南谷崇. 依存性グラフを用いた 2 相式非同期回路の合成. 信学論 (D-I), Vol. J77-D-I, No. 8, pp. 548–556, August 1994.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techinques and Tools.* Addison Wesley, 1986.

[3] 高村明裕, 桑子雅史, 南谷崇. 非同期式プロセッサ TITAC-2 の論理設計における高速化手法. 信学論 (D-I), Vol. J80-D-I, No. 3, March 1997. (掲載予定).