Communicating Logic: An Alternative Embedded Stream Processing Paradigm

Norbert Imlig, Ryusuke Konishi, Tsunemichi Shiozawa, Kiyoshi Oguri,
Kouichi Nagami, Hideyuki Ito, Minoru Inamori, and Hiroshi Nakada

NTT Network Innovation Laboratories
1-1 Hikari-no-oka, Yokosuka-shi, Kanagawa-ken, 239-0847, Japan
{imlig, ryusuke, shiozawa, oguri, nagami, hi, ina, nakada}@exa.onlab.ntt.co.jp

Abstract— This paper introduces a new concurrent processing model called "Communicating Logic" (CL). As the target architecture we adopt the dual layered "Plastic Cell Architecture" (PCA). Data-path-oriented processing functionality is encapsulated in asynchronous hardware objects with variable graining and implemented using look-up tables. Communication (i.e. connectivity and control) between the distributed processing objects is achieved by means of inter-object message passing. The key point of the CL approach is that it offers the merits of scalable, high performance hardware implementation with the compilation and linking capabilities unique to software.

I. Introduction

Although many researchers are working towards "closing the hardware-software gap" it has not completely vanished yet. In spite of the introduction of programmable logic devices (e.g. FPGAs) and sophisticated synthesis tools, reconfigurable computing is still immature. There are many reasons why people from the *stored program logic* (i.e. software) community take a rather pessimistic stance against *wired logic*. Hardware is static, hard to compile, and scalability is limited to the chip size of the reconfigurable devices used. On the other hand, hardware engineers dislike the clumsy multi-CPU processing approach because the grain size of the elements is coarse and fixed.

The *communicating logic* approach tries to bridge the inherent difference between the hardware and the software views. While software involves finding the best sequential algorithm for solving a problem cost-effectively, the hardware designer tries to find as much parallelism as possible to achieve maximum performance. The two key aspects in realizing parallel processing are concurrency and communication. Every parallel processing problem can be solved by optimizing the balance between these two properties. Concurrency is exploited by dividing a problem into independent subtasks for parallel execution or into dependent subtasks for pipelined execution. Communication is the art of connecting and controlling the processing objects.

The goal of CL is to directly interface variable grained logic circuit objects using a flexible communication network and to dynamically control the dataflow during operation. Because CL applications run on top of the asynchronous, dual-layered PCA architecture (Fig. 1), they are truly scalable. Even multiple chip configurations appear to the user as a huge array of programmable processing, memory and communication facilities. Independent processing objects are mapped onto the look-up tables (LUTs) of the *logic layer*. Since the size of these objects is modest they can be efficiently compiled or instantiated from a prepared parameterizable circuit library. The LUTs of the logic layer are also used for generating memory objects. Objects can be freely placed on the 2-dimensional LUT array because they possess no I/O terminals. The only way to communicate with other objects is by receiving/sending messages through their ports via the *communication layer*. A message can be thought of as virtual channel consisting of payload data preceded by header commands like "set the route west, north, etc." that are processed by the switches of the communication layer. A fundamental characteristic of the CL approach is the separation of processing and communication. Since objects can be freely placed on the logic layer there are no mapping and routing constraints. Object generation can be compared with the compilation process of software modules; their I/O ports are connected in the linking process. Data-path objects are accessed and controlled like subroutines.

In our opinion, an application targeted for parallel execution must be coded in a hardware-oriented language with designated syntax constructs for describing parallel tasks. In order to fully exploit the advantages of the fine grained parallelism unique to wired logic we have already developed a hardware description language called SFL (Structured Function description Language) [1]. However, SFL and the PARTHENON tools were targeted for synchronous designs. The CL approach is based on asynchronous dataflow semantics.

The rest of the paper is organized as follows. Sect.

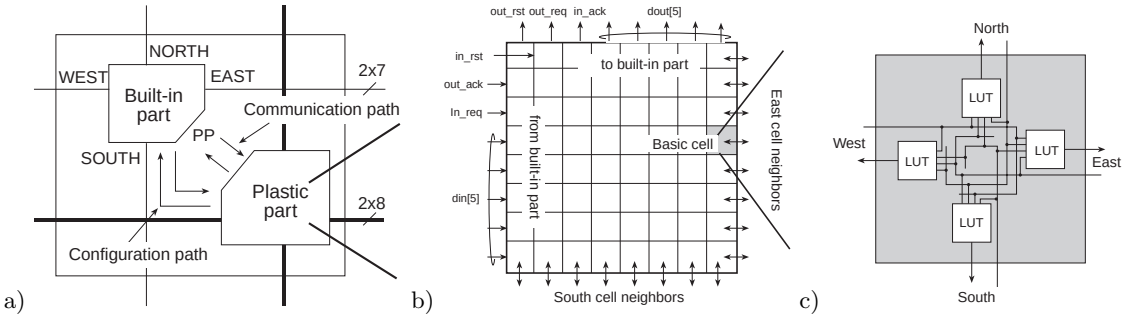

Fig. 1. a) Basic PCA cell with built-in part (communication layer) and plastic part (logic layer). b) Plastic part with interface pinout to built-in part. The data-path is 4 bits wide because the most significant bit of *din[], dout[]* is used for special signaling c) A *basic cell* corresponds to four symmetric 4-input one output LUTs for implementing logic functions, wires and asynchronous circuit elements

II gives a short overview of the Plastic Cell Architecture (PCA). We then outline the basic framework of the CL approach. Sect. IV describes the compilation and linking process. After explaining asynchronous circuit synthesis in Sect. V, we conclude by pointing out some considerations about future work.

II. PCA architecture

Fig. 1-a shows the basic cell of the PCA architecture [2, 3, 4]. We call the variable part of the logic layer the *plastic part* and the routing switches of the communication layer the *built-in part*. In order to exploit parallelism, the cells are repeated and arranged in an orthogonal array. All I/O enters the processing array encapsulated as messages via the built-in part. There are two types of interconnection paths between the built-in part and the plastic part. The *configuration path* is used for writing and reading the LUT memory array. The *communication path* is for inter-object message passing. A gate in the communication path can be opened/closed by a PCA command. In the initial state, all gates are closed and the corresponding object cannot send or receive any message. This is because we do not want any random events while an object is being constructed.

A. Built-in part

The switches of the built-in part handle message routing and the configuration of the plastic part LUT memory. An asynchronous state machine interpreting the PCA commands exists at every input port. Except in the case of an output port conflict, all input ports behave independently. Arbiters are responsible for smooth asynchronous operation. The valid commands for routing and the syntax of the basic message format are explained in [5]. Similar to the UNIX philosophy, whose clean interface is based on 4 constructs *open, read, write* and *close*, our message alphabet is very simple. The write and read messages configure/read the LUT memory through the configuration path. The copy message is for fast reproduction of on-chip circuit templates. The data message is for exchanging information between circuit objects via the communication path. The pinout of the interface to the plastic part is shown in Fig. 1-b. After receiving an *OPEN* command, the neighboring connectors to the west and north of the plastic part are switched to the built-in part.

B. Plastic part

Every built-in part controls a cluster of 8x8 basic cells. Fig. 1-c shows the structure of a single cell. It consists of four symmetric 4-input one output LUTs. The LUT array can also be used as a 1024 4-bit wide memory object. The most important feature of the plastic part is its regular orthogonal structure which allows dynamic circuit allocation [6]. While conventional FPGAs consist of several building blocks like look-up tables, routing switches, and flip-flops, our plastic part is very simple. Logic functions including wires are all implemented using basic LUT elements. Due to the asynchronous circuit paradigm there is no global clock signal and also latches and asynchronous circuit elements are constructed using feedback coupled LUTs. Thus, the resources of the plastic part can be freely balanced when implementing logic, wires and asynchronous circuit elements. This is similar to the software case where code (instruction, data) as well as communication (pointer references) can be freely interchanged.

III. CL framework

As stated in the previous section, the key aspect of the CL approach is the separation of processing and communication. A theoretical framework for distributed parallel computing can be found in Hoare's communicating sequential processes (CSP) [7] in which concurrent processes communicate over fixed channels. Channels are used for synchronization and data exchange. In the CL framework a channel corresponds to a *message* and a process to a wired logic *circuit object*. A message is a stream of commands forming a virtual channel to an object followed by a variable amount of data samples used for processing.

Messages can be generated by circuit objects or are injected via I/O ports from outside.

The main difference between CSP and our approach is the fact that in the CSP framework, channels, as well as objects, must be fixed at compile time. On the other hand, the dynamic properties of the PCA architecture make it possible to freely allocate CL objects and channels at run-time.

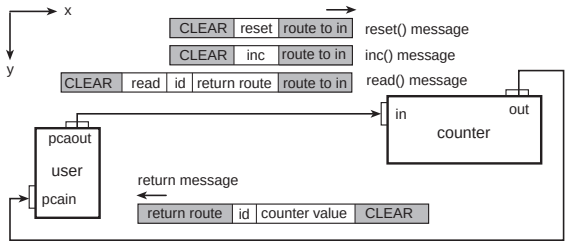

Fig. 2. Object-message relationship of simple *counter* object and *user* control object

```
 1 OBJECT counter {
 2   INPORT in;
 3   OUTPORT out;
 4   MESSAGE in::reset(), in::inc(), in::read();
 5   REG count;
 6
 7   RECEIVE in ALT {
 8     reset: count:= 0x0;
 9     inc:   count++;
10     read:  SEND out(count);
11     else:  SEND out(in);
12   }
13 }/*counter*/
```

```
 1 #include <iostream.h>
 2 #include "pca.h"
 3 #include "counter.h"
 4
 5 OBJECT user {
 6   INPORT pcain;
 7   OUTPORT pcaout;
 8   REG count, id;
 9   counter *c[2];
10
11   for (int i=0; i<2; i++) {
12     c[i]= new(counter);
13     pcaout << c[i]->reset();
14   }
15   for (int i=0; i<5; i++) {
16     pcaout << c[0]->inc();
17     pcaout << c[0]->read(pcain, 0);
18   }
19 }/*user*/
```

Fig. 3. Code of hardware processing object *counter* and software control object *user*

A. Objects and messages

An abstract specification of the behavior of an object can be expressed by its message interface description. Fig. 2 depicts the object-message relationship of a simple counter circuit with one input and one output port. This object can be only accessed from the outside using the messages reset(), inc(), read(), and thus be placed anywhere on the 2-dimensional cell array. The code shown in Fig. 3 describes the counter object and a possible user object. Because of the asynchronous handshaking protocol explained in Sect. V, a processing object only makes a transition when its input- and output ports are ready. This synchronization is expressed using the keywords *RECEIVE* and *SEND*.

The counter object is set to zero with the reset() message. This message can be divided into a header consisting of routing commands from the user to the counter object and a payload data instruction *reset*. The information about the relative position of the two objects is stored in a system wide allocation table, which is updated each time a new object is generated or deleted. The *CLEAR* command at the end of the message releases the route set by the header commands and the path resources are freed for other messages. The counter clears its count register upon receiving the reset instruction. The inc() message increments the counter by one. If the user object wants to read the contents of the count register, it sends a read() message consisting of the return route, an identifier and an instruction to read the count register. The counter passes the return route and the identifier to its output port before appending the contents of the count register.

The uniform message interface makes it easy to directly access hardware objects by software calls. For example in Fig. 3 the user object is implemented as an embedded software object by using the library functions of the counter object class. Every function call returns the appropriate message stream, which is then injected, via an I/O port *pcaout*. When the object is generated with the command *new(counter)* the counter class constructor injects a configuration message with the circuit information as payload into the logic layer.

B. Stream-processing

All objects process streams of data. A stream is the basic unit of processing and can have arbitrary length. Memory objects may contain data for processing or message information for the control of processing objects. In terms of communication there are two types of objects. *Master objects* are able to actively send messages while *slave objects* only process and forward messages to the next processing stage. If necessary, slave objects acknowledge the termination of processing to the initiating master object. The counter object in Fig. 2 is a slave object while the user object is its master object. Fig. 4 shows the basic building blocks for controlling the dataflow of streams. All input and output data transmissions are based on an asynchronous 4-cycle bundled data protocol.

The *pipeline composition* is the key configuration for achieving high throughput performance. Basically there are two ways to connect the pipeline stages. In the static

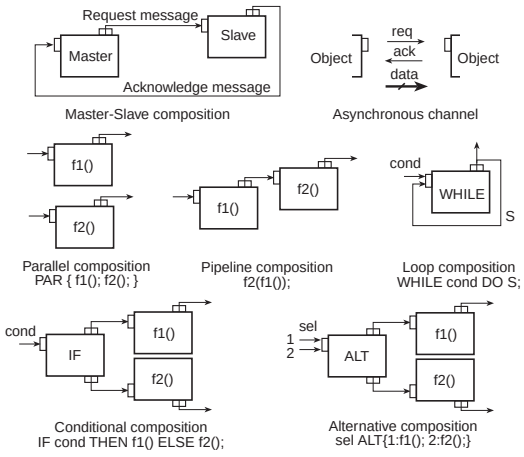

Fig. 4. Basic building blocks for controlling the dataflow of streams

path scheme, a master object opens a static path through all pipelined slave objects before streaming the actual processing data to them. In the dynamic path scheme a communication stub is inserted by the CL linker that forwards the processed data encapsulated in a message to its neighboring object.

The *loop statement* recursively reads out a memory object that contains the control commands of the body S as long the condition is true. Also subroutines can be realized in this way. The *conditional composition* allows stream manipulation based on the evaluation of a boolean variable.

The *alternative composition* introduces the need for arbitration. Since the resources of the logic layer are limited, arbitration is only carried out in the switches of the communication layer. Thus the logic and the communication layer share the burden resulting from asynchronous implementation which means that the message streams must use the same input port.

IV. Compiler and linker

The main motivation behind the automated translation of programs is to improve efficiency and correctness. Syntax-directed compilation guarantees the correct translation from upper to lower levels. In Sect. III we introduced the two key properties needed when describing circuits with the CL approach:

- Timing independence
- Location independence

These properties are critical for the success of a high level programming language. Independence of timing results from the asynchronous processing model. The data-driven self-timed approach replaces the requirement of correct timing by correct sequencing and, most importantly, allows vast hardware scalability.

Independence of location allows compilation to be separated from the linking process. Compilation corresponds to the separate generation of circuit objects while linking can be compared to the generation of messages for connecting these objects. Thus the compilation process of CL strongly resembles that of software. At the moment, the compilation of hardware objects is based on a library approach. Parameterizable circuit macro-modules are laid out by hand. The compiler constructs objects by transforming prepared circuit templates. Since CL objects are thought to be small, this is not a severe limitation. After CL synthesis, the circuit information is translated into injection files with the LUT configuration data. The CL linker resolves missing connections by patching the injection files with communication stubs. It is also responsible for realizing deadlock free, well-balanced message routing on the communication layer.

V. Asynchronous circuit synthesis

In this section we explain the asynchronous synthesis framework following the work in [8, 9, 10]. We have chosen asynchronous implementation for the following reasons:

1. Variable graining: Due to the dynamic nature of PCA, objects of different size can be instantiated at run-time. It would be difficult to adjust the clock cycle after a new object has been allocated.

2. Scalability: It is difficult to distribute a large scale clock signal on- and off-chip without clock skew and peak power consumption problems.

Among the disadvantages of asynchronous circuits we note that they are more complex to design than their synchronous counterparts due to the handshake protocols. Further they lack an accepted methodology to verify and test them. Because the asynchronous handshaking protocol of PCA must be implemented on its logic layer by using LUTs, an efficient implementation is important.

A. Handshaking protocol

Fig. 5 shows the basic handshake protocol between a sender and a receiver circuit. We chose the *4-cycle bundled data* protocol [11] to control data transmission. We prefer this method over the *dual-rail encoding* protocol [12] because dual-rail encoding needs twice as many wires (cells) and complicated circuits for completion detection. Furthermore, the bundled data protocol suits the layout requirements of the PCA architecture. Since logic and wire connections are all realized with LUT elements, delays can be estimated accurately. Finally, circuits implemented with the 4-cycle protocol tend to be smaller than those realized with the 2-cycle protocol. For example a D-latch can be efficiently realized with two LUTs of neighboring basic cells as depicted in the lower part of Fig. 6-b.

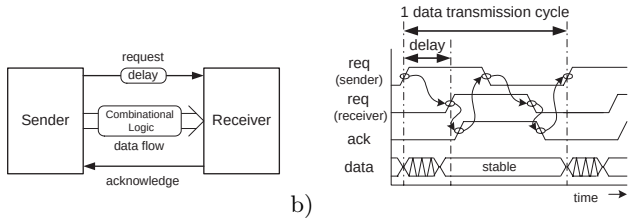

Fig. 5. a) Basic asynchronous handshaking b) Timing diagram of 4-cycle bundled data protocol

Fig. 5-b shows the timing diagram of the 4-cycle bundled data handshaking protocol. The delay in the request signal path assures that the data has passed through the combinational logic circuit when it reaches the receiver. Thus we do not have to deal with hazards in the data-path and combinational circuits can be generated using common logic synthesis tools.

This timing diagram can be realized with the circuit shown in Fig. 6-a. The specification of the Muller-C element is described by the truth table shown in Fig. 6-b. The D-latch is very similar to that used in synchronous circuits. The additional acknowledge signal C' is required for approving the stabilized latch output.

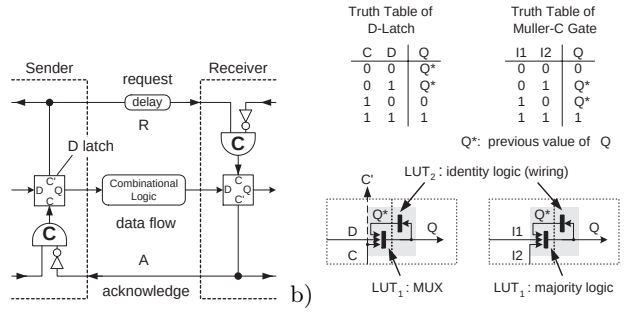

Fig. 6. a) Implementation of 4-cycle handshaking circuit. The gate labeled 'C' is the Muller-C element. b) Truth table and LUT realization of D-latch and Muller-C element

B. General handshaking configurations

Fig. 7 shows several circuits for controlling pipelined data-paths. The *fork* circuit sends data transfer requests to multiple receivers at the same time, and waits until all receivers have returned their acknowledge signals. In the case of a *join* circuit, a receiver waits for all requests from multiple senders, and acknowledges them simultaneously. A *selector* circuit switches the handshake signals of two receivers according to the data received. For correct timing, the handshaking signals must be switched during the delay period indicated in Fig. 5-b.

The *loop* handshaking circuit shown in Fig. 7-d can be used to construct a finite state machine (FSM). A 4-cycle asynchronous implementation of a loop needs at least three latch stages. Two for the master and slave stages and an additional buffer latch before feeding back the outputs.

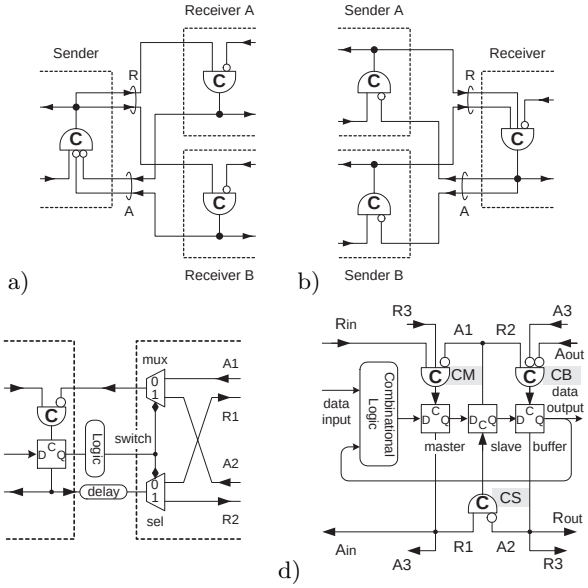

Fig. 7. Handshaking circuits for controlling pipelined data-paths a) Fork b) Join c) Selector and d) Loop

C. Layout of counter object

Fig. 8 shows the block diagram of the counter circuit introduced in Sect. III. The message data arrives from the built-in part on the left side of the figure and gets decoded. The reset() and inc() messages control the counter subcircuit realized as 4-bit slices of a basic FSM. The return route and identifier of the read() message are forwarded through the MUX via the buffer to the output port. If the read() command is decoded, the MUX switches the data-path from the input port to the counter FSM in order to forward the counter value. Three Muller-C elements control the latches. Because the reset() and inc() messages handshake with the counter FSM, while the read() message handshakes with the output port, a handshake selector circuit is needed. If the output port is not involved in handshaking, the corresponding request and acknowledge signals are shortcut.

D. Initialization

As with synchronous designs, asynchronous circuits must also be initialized for proper operation. This is achieved by the OPEN command before connecting the plastic part to the built-in part. All the D-latches and Muller-C elements are connected in a daisy-chain manner. The start point of the chain is the *in_rst* pin and the end point the *out_rst* pin as shown in Fig. 1-b. A low to high transition forces the asynchronous circuit elements

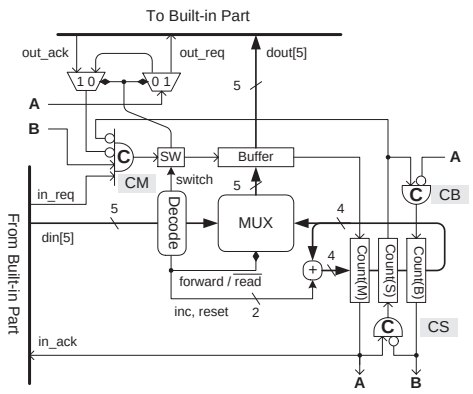

Fig. 8. Block diagram of counter object. Rectangles correspond to latches, while rounded rectangles symbolize combinational logic.

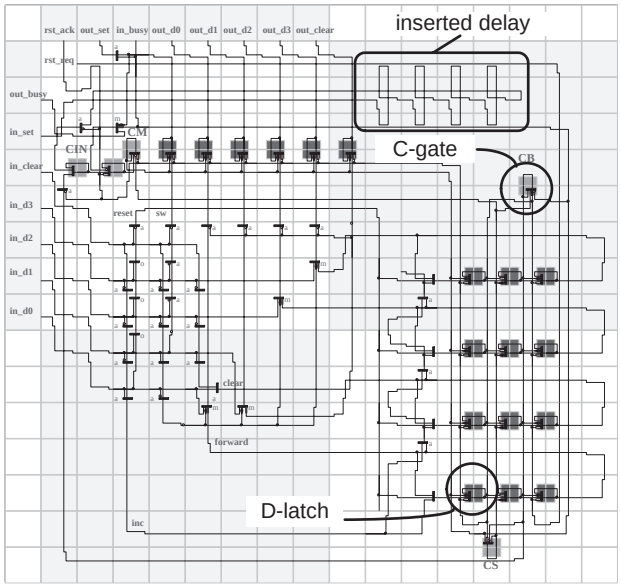

Fig. 9. Final layout of counter object realized with 16x16 basic cells (2x2 PCA cells).

into their initial state. The operation of the object starts after a high to low transition.

Fig. 9 shows the final layout of the counter object. The inserted delay in the request path implemented by zigzag wiring assures correct timing. We have simulated this circuit on our asynchronous simulator tool *PCASIM* and verified its behavior.

VI. Conclusion

We have introduced a new concurrent processing paradigm based on asynchronous programmable logic. By strictly separating processing (circuit objects) from communication (messages), the communicating logic approach allows the compilation and linking steps to be separated, similar to software, which is critical for the success of a high level description language. We are now generating an asynchronous circuit library for signal processing, database, and cipher applications. A VLSI prototype chip (0.25 μm CMOS process technology) will be available in the first quarter of 2000. In our opinion, this device will obsolete today's CPU-oriented embedded architectures and be an important step towards closing the software-hardware gap.

References

[1] Y. Nakamura, K. Oguri, and A. Nagoya: "Synthesis From Pure Behavioral Descriptions," High-Level VLSI Synthesis, Edited by R. Camposano, and W. Wolf, Kluwer Academic Publishers, pp.205-229, June, 1991
`http://www.kecl.ntt.co.jp/parthenon/`

[2] T. Shiozawa, K. Oguri, K. Nagami, H. Ito, R. Konishi, and N. Imlig: "A Hardware Implementation of Constraint Satisfaction Problem Based on New Reconfigurable LSI Architecture," in Proc. of FPL'98, Springer Verlag, LNCS 1482, pp.426-430, August, 1998

[3] K. Oguri, N. Imlig, H. Ito, K. Nagami, R. Konishi, and T. Shiozawa: "General Purpose Computer Architecture Based on Fully Programmable Logic," in Proc. of ICES'98, Springer Verlag, LNCS 1478, pp.324-334, September, 1998

[4] H. Nakada, K. Oguri, N. Imlig, M. Inamori, R. Konishi, H. Ito, K. Nagami, and T. Shiozawa: "Plastic Cell Architecture: A Dynamically Reconfigurable Hardware-based Computer," in Proc. of IPPS/SPDP'99, Springer Verlag, LNCS 1586, pp.679-687, April, 1999

[5] N. Imlig, T. Shiozawa, K. Nagami, R. Konishi, and K. Oguri: "Communicating Logic: Digital Circuit Compilation for the PCA Architecture," in Proc. of DA Symposium'99, IPSJ Symposium Series Vol.99, No.8, pp.101-106, July, 1999

[6] K. Nagami, K. Oguri, T. Shiozawa, H. Ito, and R. Konishi: "Plastic Cell Architecture: Towards Reconfigurable Computing for General-Purpose," in Proc. of FCCM'98, pp.68-77, April, 1998

[7] C. A. R. Hoare: "Communicating Sequential Processes," Communications of the ACM, 21, pp.666-677, 1978

[8] I. E. Sutherland: "Micropipelines," Communications of the ACM, 32, 6, pp.720-738, June, 1989

[9] K. van Berkel: "Handshake Circuits," Cambridge University Press, ISBN 0-521-45254-6, 1993

[10] T. H. Meng: "Synchronization Design for Digital Systems," Kluwer Academic Publishers, ISBN 0-7923-9128-4, 1991

[11] C. L. Seitz: "System Timing," in C. Mead, and L. Conway, "Introduction to VLSI Systems," Addison-Wesley Publishing Company, ISBN 0-201-04358-0, pp.218-262, 1980

[12] D. B. Armstrong, A. D. Friedman, and P. R. Menon: "Design of Asynchronous Circuits Assuming Unbounded Gate Delays," IEEE Transactions on Computers, C-18, 12, pp.1110-1120, December, 1969