© 2006 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. Digital Object Identifier : 10.1109/ISIE.2006.296082

Converting PLC instruction sequence into logic circuit: A preliminary study

Shuichi Ichikawa*†, Masanori Akinaka*, Ryo Ikeda*, and Hiroshi Yamamoto*
* Dept. Knowledge-based Information Engineering, Toyohashi University of Technology
1-1 Hibarigaoka, Tempaku, Toyohashi 441-8580, Japan
† Intelligent Sensing System Research Center, Toyohashi University of Technology
1-1 Hibarigaoka, Tempaku, Toyohashi 441-8580, Japan
Email: ichikawa@tutkie.tut.ac.jp

Abstract— **By implementing a control program with hard-wired logic using reconfigurable devices (e.g., FPGA), a flexible and highly responsive system can be realized. This new system also contributes to securing intellectual property, while reducing implementation space and cost. This study outlines a converter that translates PLC instruction sequence into logic description. A design framework is also described, which integrates control logic and peripheral functions on an FPGA chip. A productive ladder program was examined with Mitsubishi Electric FX2N PLC and Altera APEX20KE FPGA, and the derived logic designs were shown to fit into an actual FPGA chip. A straightforward Sequential design was estimated to be 79 times faster than PLC, while a performance-oriented Flat design was estimated to be 43 times faster than Sequential design (i.e., 3397 times faster than PLC).**

I. Introduction

A Programmable Logic Controller (PLC) is a kind of computer, which has been widely adopted for sequence control of industrial machinery. Though PLC is flexible and well-established, the performance of PLC does not always satisfy the requirements in large and highly responsive systems. Another problem of PLC is that a PLC program is easy to duplicate and to analyze. This often results in the leakage of valuable trade secrets and the rise of clone products.

By implementing a control program with hard-wired logic using reconfigurable devices (e.g., FPGA), a flexible and highly responsive system could be realized. Since an FPGA chip can contain maximally ten million logic gates, a very large control system could be implemented with a single chip. This may sometimes lead to downsizing and reduction of system components. It should be also noted that FPGA is more secure than PLC in protecting intellectual properties, because (1) it is more difficult to analyze an FPGA design than to analyze a PLC program, and (2) some recent FPGA devices provide design security features (e.g., Altera Stratix-II).

It is evident that there are some drawbacks in adopting FPGA. For example, circuit generation time is consumed whenever the control program is updated. Reliability and noise immunity issues are also practical concerns. The authors *never* insist on replacing all conventional PLCs with FPGAs. Rather, we suggest that FPGA technology might offer a promising alternative solution for some applications, particularly for highly responsive systems.

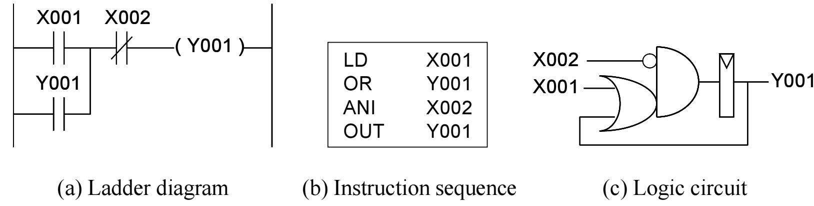
This study presents an experimental converter, which translates PLC instruction sequence into logic description in VHDL [1]. It also introduces the implementation of a control logic library, which includes various components to support PLC instructions and peripheral devices. A design framework is then outlined, which integrates control logic and peripheral functions into an FPGA chip.

II. Background and Related Studies

There have been some studies on the implementation of a control program in FPGA. For example, Adamski and Monteiro [2][3] presented a design methodology that translates “interpreted Petri net specification” into hardware description languages. Wegrzyn et al. [4][5] presented a framework that transforms rule-based descriptions (e.g., interpreted Petri net) into logic descriptions (e.g., VHDL). Ikeshita et al. [6] presented a conversion program that translates SFC (Sequential Function Chart) description into Verilog-HDL for logic synthesis. All these studies concentrated on techniques to convert functional-level control programs into logic circuits. In contrast, this study deals with control programs at the lowest level: PLC instruction sequence. Although it is more difficult to analyze, our scheme would be applicable to a wider area of control programs. Moreover, our technique might be extendible to the instruction sequence of various embedded processors.

Miyazawa et al. [7] proposed a method to translate PLC programs of ladder diagram into VHDL programs. Welch and Carletta [8] proposed an FPGA architecture, which implements relay ladder logic directly. Though the ladder diagram is almost equivalent to a PLC instruction sequence, they only examined very fundamental logic functions such as AND, OR, NOT, and flipflop, while providing no detailed discussion about actual PLC applications. The present study, however, deals with advanced features of PLC that are required in real-world applications.

The ladder diagram has been widely accepted to describe PLC programs. A ladder diagram consists of one or more *rungs*, each of which consists of a condition part and a process part (Fig. 1). Either the condition part or the process part can be an input/output (a) or an instruction (b). The output of a rung is activated if the corresponding input condition is



(a) Ladder diagram (b) Instruction sequence (c) Logic circuit

Fig. 2. An example of FX2N PLC program: self-holding logic.

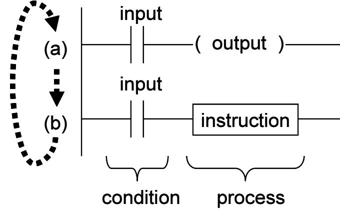


Fig. 1. Overview of ladder diagram.

satisfied; otherwise, the output is deactivated. The instruction of a rung is executed if its input condition is satisfied. Rungs are ordered, and interpreted in due order.

A ladder diagram is executed in the following manner:

1) At the beginning of a ladder, all inputs are collected and stored into the corresponding internal memory elements, which are read and modified by rungs (Input phase).
2) Rungs are interpreted in due order (Execution phase).
3) When the bottom of a ladder is reached, all output ports are updated by the corresponding internal memory values (Output phase).
4) The ladder is then executed all over again from the input phase.

Repeated execution of the abovementioned cycles is called *cyclic scan*, and the period of cyclic scan is called *scan time*. By making the scan time shorter, the system becomes more responsive.

III. Translation of PLC program to hardware description

A. Translation of a Rung

Figure 2 (a) illustrates a rung of a ladder diagram for Mitsubishi Electric FX2N PLC [9], which is adopted as an evaluation platform in the following discussion. A distinct advantage of FX2N PLC is that its instruction set specifications are open to the public [10].

FX2N instruction set includes 160 instructions with various types of operands: e.g., switch X, coil Y, internal relay M, data register D, constant K, and timer T. In Fig. 2, the switches X001 and X002 correspond to start switch and stop switch, respectively. The slash on X002 denotes negative logic. If X001 is on and X002 is off, the output coil Y001 is turned on; this results in X001 bypassed, and thus Y001 holds while X002 is off. If X002 is turned on, Y001 is inactivated. This logic is called *self-holding logic*. The rung (a) is translated into the instruction sequence shown in Fig. 2 (b), while this control logic can be translated into the corresponding logic circuit (Fig. 2 (c)).

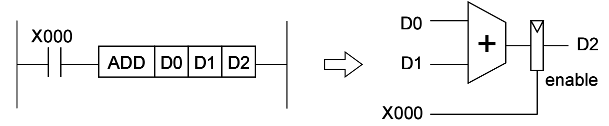


Fig. 3. Another example: arithmetic instruction.

Another example is shown in Fig. 3, where the process part of a rung is an arithmetic instruction. In this case, an ADD instruction is translated into an adder, whose output is captured by a register if the corresponding condition is satisfied.

Our experimental converter generates a VHDL source code from an instruction sequence of FX2N. The supported instructions include 16 basic programming instructions (out of 27) and 5 applied instructions (out of 133), which are summarized in Table I. This list is not long, but includes enough instructions for the following evaluations. The authors are still extending the list of supported instructions, basically on a demand-driven basis to support real-world control programs.

B. Sequential Design

To generate a logic circuit that literally simulates a whole ladder program, it is straightforward to design a sequential circuit, which activates one rung for each cycle in due order. This design is designated by *Sequential* design in the following discussion.

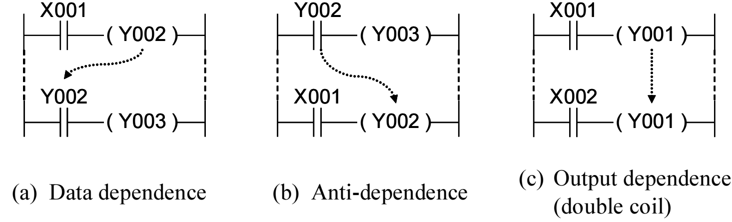
Although Sequential design reproduces the exact behavior of a ladder program, it requires $\rho + 2$ cycles for each scan. Here, ρ is the the number of rungs of a ladder, and two additional cycles are required for input phase and output phase described in Section II. For further reduction of scan time, it is essential to utilize parallelism in the control program.

C. Levelized Design

It is possible to execute two or more rungs in parallel, as a superscalar microprocessor does, if dependences among rungs are properly maintained. Figure 4 (a) shows an example of data dependence, where the output of the upper rung is referred by the lower rung. In Fig. 4 (b), the input of the upper rung is overwritten by the lower rung (anti-dependence). Figure 4 (c) shows an example of output dependence, or double coiling [9], where the output of the upper rung is overwritten by the lower

TABLE I
SUPPORTED INSTRUCTIONS.

Category	Mnemonic
Basic instructions	LD, LDI, AND, ANI, OR, ORI, ANB, ORB, INV, NOP, END, OUT, SET, RST, PLS, PLF
Applied instructions	MOV (DMOV), ADD (DADD), SUB (DSUB), MUL (DMUL), DIV (DDIV)



(a) Data dependence (b) Anti-dependence (c) Output dependence (double coil)

Fig. 4. Three dependences to be considered.

rung. In any of these cases, it is essential to execute the upper rung before the lower rung to derive the same result as in the original ladder diagram.

Dispatching each rung to the earliest cycle possible (while keeping all dependencies), we can reduce the clock cycles required for each scan. In this study, we simply levelize rungs according to their dependencies, as in levelized compiled code simulation of a logic circuit [11]. The rungs that have no preceding rungs are labeled by $Level_1$, and the rungs that are dependent on $Level_i, Level_j, \dots$ are labeled by $Level_{\max(i,j,\dots)+1}$. We can emulate a ladder by a sequential circuit, which activates the input phase at $Cycle_0$, the rungs of $Level_i$ at $Cycle_i$, and the output phase at $Cycle_{\lambda+1}$, where λ is the maximal level of rungs. Thus, the number of cycles for each scan would be $\lambda + 2$ in this circuit. This design is designated by *Levelized* design in the following discussion.

D. Flat Design

Sequential design activates the circuit of each rung, one for each cycle, in due order. Levelized design activates the circuit block of each level, one for each cycle, from upstream to downstream. This brings up the question of why it is implemented by a sequential logic circuit at all. It is not necessary to split the execution phase into cycles, because the inputs and outputs are updated only at the end of each scan.

In fact, it is possible to implement the execution phase by a combinatorial logic circuit. Let us examine Levelized design as an example. The input data of $Level_j$ circuit are fed from $Level_i$ $(0 \le i < j)$ in Levelized design. When the process part of a rung at $Level_j$ is an output, we can simply remove the internal memory element of this output, and feed data downstream by wire. When the process part is an arithmetic instruction, we have to replace the memory element by a multiplexer, which feeds data downstream. Since the instruction might or might not take place depending on the value of its condition part, a multiplexer is required to select either the new value (generated at $Level_j$) or the original value (fed from $Level_i$). The execution phase could thus be converted into a combinatorial logic circuit.

The input and output phases are also redundant. In Sequential and Levelized designs, two cycles are consumed for input and output phases. Since the inputs are always updated just after the outputs, the input phase and output phase can be unified to one cycle.[1]

Taking these two ideas together, each scan could be performed in one cycle. The derived design is designated by *Flat* design in the following discussion. Flat design is expected to be faster than Levelized design for the following reason. The scan time of Levelized design t_l is given by $t_l = (\lambda + 2)\max_i \delta_i$, where δ_i is the maximal delay of $Stage_i$ (the circuit of $Level_i$). Assuming that the difference in logic is negligible, the scan time of Flat design t_f is expected to be shorter than t_l, because $t_f \le \Sigma_i \delta_i \le \lambda \max_i \delta_i < t_l$ holds. In many cases, Flat design would be much faster than Levelized design, because $t_f \ll \Sigma_i \delta_i$ usually holds. Another advantage of Flat design is that CAD software is generally good at optimizing combinatorial logic circuits, compared to sequential circuits.

E. Resource Restriction

Though scan time is very important, the logic scale of control circuit is equally important for practical applications. Particularly in translating a large control program, it is essential to restrict resource usage. In this section, we discuss the circuit generation with resource restriction.

In the abovementioned designs, each instruction is translated into its hardware counterpart, and the consequent circuit contains as many components as instructions. A logic operation does not cost much, because it is implemented by a bitwise circuit. Meanwhile, an arithmetic instruction matters very much, because it requires a 16-bit or 32-bit wide arithmetic unit. Thus, it is very important to restrict the number of arithmetic units by sharing them among instructions.

[1]Strictly speaking, the unification of input phase and output phase may cause trouble, wherever there exists an external loopback from output to input. Such design is rather exceptional, and should be avoided to assure high performance.

TABLE II

EVALUATION RESULTS OF A PID CONTROLLER PROGRAM.

Device	Design	Arithmetic unit	Num. of states	Max. Freq. [MHz]	Logic Scale [LE]	Scan time [s]	AT product [LE · s]
PLC	–	–	–	–	–	7.98×10^{-4}	–
FPGA	Sequential	dedicated	13	32.33	2459	4.02×10^{-7}	9.89×10^{-4}
		shared $\times 1$	20	29.06	1812	6.88×10^{-7}	1.25×10^{-3}
	Levelized	dedicated	8	32.69	2449	2.45×10^{-7}	6.00×10^{-4}
		shared $\times 1$	16	30.10	1744	5.32×10^{-7}	9.28×10^{-4}
		shared $\times 2$	14	31.75	2639	4.41×10^{-7}	1.16×10^{-3}
		shared $\times 3$	12	30.63	3403	3.92×10^{-7}	1.33×10^{-3}
	Flat	dedicated	1	20.04	3145	4.99×10^{-8}	1.57×10^{-4}

In Sequential design, it is very easy to share arithmetic units, because only one rung is activated in each cycle. It is enough to generate one arithmetic unit for each kind of arithmetic operation, if its input is multiplexed and its output is redirected properly. In the following discussion, this design is designated by *shared* arithmetic units. It should be noted that a shared design might incur a significant amount of hardware for the input multiplexers and output interconnects in exchange for reduction of arithmetic units. The original design, which generates as many arithmetic units as arithmetic instructions, is designated by *dedicated* in the following discussion.

In Levelized design, resource limitations of an arithmetic unit may affect the scheduling of rungs, which can result in the increase of clock cycles. It is thus necessary to choose a good resource scheduling algorithm to minimize the scan time. This problem is a type of multiprocessor scheduling problem [12][13] which is generally difficult to solve. Though a simple ASAP scheduling was implemented in this study, better algorithms should be investigated in future studies.

In our particular implementation of shared arithmetic units, we allocated one additional cycle to each arithmetic operation for data transfer. This is an implementation-specific issue, and generally not essential.

F. Optimization Issues

Though there are many possible optimizations, we left most of them for future studies, and concentrated on evaluating the fundamental aspects of this method. The following are some items for future attention.

The first issue is the optimization of instruction sequence. In this study, our converter literally translates an instruction sequence into the corresponding logic description. However, it is possible to generate a better logic circuit by analyzing and rewriting the instruction sequence. For example, the sum of four values a, b, c, d can be calculated by either $((a+b)+c)+d$ or $(a + b) + (c + d)$. Literally converted, the former would result in a cascade of three adders, while the latter would be a balanced tree of three adders. Generating a balanced tree of adders from the former instruction sequence is left for future studies.

Area-Time trade-off is another important issue. Although some results are shown in the following evaluation, automatic exploration of the best trade-off is beyond the scope of this work.

IV. EVALUATION RESULTS

This section presents some evaluation results of two sample PLC programs. The evaluation flow is shown by broken lines in Figure 5. First, the scan time of PLC (H) is estimated from its instruction sequence (B) according to the execution time for each instruction [9]. Since the execution time of each instruction is dependent on the value of the corresponding condition part, the worst case scan time is estimated in this evaluation. The PLC instruction sequence is then translated into the hardware description in VHDL by our translator (C). This VHDL description is processed by Altera Quartus II 4.0 software to generate an FPGA design for Altera APEX20KE FPGA [14]. In this study, the target device was set to EP20K600E with 24320 LE (logic elements). The optimization options of Quartus II are set to default. The scan time of FPGA (I) is estimated by the estimated maximum operational frequency and the number of states of the circuit.

Table II summarizes the evaluation results of a PLC program which implements a simple PID controller with 32-bit fixed point arithmetic. The PID controller is classical, but has been frequently used in many control applications to date. This PLC program includes 23 instructions, which include 4 add/subtract instructions and 3 multiply instructions.

FX2N PLC takes 798 μsec for each scan of PID code, in which 508 μsec is consumed by END instruction. Since the END instruction is placed to finish the current scan and to carry out the process of updating outputs and inputs, 64% of PLC scan time is consumed to update inputs and outputs. In contrast, FPGA implementations can perform input/output phases in one or two cycles, which results in very high responsiveness.

Sequential (dedicated) design is about 2000 times faster than FX2N PLC, yet requires only 10% of the LEs of a EP20K600E device. Levelized (dedicated) design is 1.64 times faster than Sequential (dedicated) design, although the logic scale is almost the same. Flat design is 4.9 and 8.1 times faster than the Levelized (dedicated) and Sequential (dedicated) design, respectively.

Table II includes the results of Area-Time product (AT product) of each design, i.e., the product of logic scale and scan time. Since a smaller AT product means that the circuit requires a smaller logic scale for the same performance, it is a popular measure of cost-effectiveness. From AT product, it is

TABLE III
EVALUATION RESULTS OF SAMPLE LADDER PROGRAM.

Device	Design	Arithmetic unit	Num. of states	Max. Freq. [MHz]	Logic Scale [LE]	Scan time [s]	AT product [LE · s]
PLC	–	–	–	–	–	1.61×10^{-3}	–
FPGA	Sequential	dedicated	74	3.63	9733	2.04×10^{-5}	1.99×10^{-1}
		shared ×1	101	2.84	4565	3.56×10^{-5}	1.63×10^{-1}
	Levelized	dedicated	12	3.64	9613	3.30×10^{-6}	3.17×10^{-2}
		shared ×1	33	2.76	4523	1.20×10^{-5}	5.43×10^{-2}
		shared ×2	25	2.93	6616	8.53×10^{-6}	5.64×10^{-2}
		shared ×3	23	2.97	8007	7.74×10^{-6}	6.20×10^{-2}
		shared ×4	21	3.06	9703	6.86×10^{-6}	6.66×10^{-2}
	Flat	dedicated	1	2.11	8887	4.74×10^{-7}	4.21×10^{-3}

readily seen that Flat design is 3.8 times more cost-effective than the Levelized (dedicated) design.

In Table II, "shared $\times n$" designates a design that includes maximally n arithmetic units for each kind of arithmetic operation. Though it is possible to set a different limitation for each operation (e.g., 2 for adder and 1 for multiplier), we applied the same limit to all kinds of arithmetic operations in this experiment. Sequential (shared ×1) design achieved 26% reduction of LEs, while the reduction was 29% in Levelized (shared ×1) design. Levelized (shared ×2 and ×3) designs were larger in logic scale and slower in scan time than dedicated design. As stated in Section III-E, shared designs require more cycles than dedicated designs for each scan. This might be one of the reasons why shared designs are slow in this evaluation.

Table III lists the evaluation results of a sample PLC program, which was derived from an actual product. This PLC program includes 165 instructions, which include 6 add/subtract instructions, 12 multiply instructions, and 9 divide instructions.

Sequential (dedicated) and Levelized (dedicated) designs are almost the same in maximum operational frequency and logic scale, while Levelized (dedicated) design is 6.2 times faster than Sequential (dedicated) design. Flat design is even 7.0 times faster than Levelized (dedicated) design, yet its logic scale is 7.6% smaller. Compared to PLC, Flat design achieves 3397 times higher performance, using only 37% LEs of an EP20K600E device.

In this program, the effect of shared arithmetic units was larger than in the PID controller. Sequential (shared ×1) achieved 53% reduction of LEs, while providing 18% better AT-product. Levelized (shared ×1) design also achieved 53% reduction of LEs, though its AT-product was 71% less than Levelized (dedicated) design. Levelized (shared ×2 and ×3) were also smaller than Levelized (dedicated) design, although slower.

V. DESIGN FRAMEWORK

Figure 5 illustrates the framework of tools available to translate, integrate, and implement the logic circuit of a control system onto a FPGA. In Fig. 5, double rectangles designate three tools implemented by the authors. The solid-line arrows designate the path to generate logic circuit from control logic, while the dotted-line arrows designate the path for performance evaluation.

A ladder program is designed with Mitsubishi GX Works software (step A in Fig. 5), and then translated into an instruction sequence with GX Converter (B). As described in Sect. III, our translation tool translates the instruction sequence into the hardware design described in VHDL (C). Users may write the top-level design (F) by themselves, or may prepare the interface description file (E) instead. The top-level design can be generated from the interface description by our interface logic generation tool. In this example, the top-level design includes a library component STPG, which is a peripheral device of FX2N PLC.

Even if a PLC program could be translated into hardware, it does not work without peripheral devices. To achieve a higher level of integration, it is essential to integrate peripheral devices on an FPGA together with the control logic circuit. Thus, the authors prepared a control logic library, which includes (1) various components to replace peripheral devices of PLC, (2) template circuits that correspond to PLC instructions, and (3) some support functions. In this example, a programmable pulse generator STPG (D) is extracted from the peripheral library. Finally, Altera Quartus II 4.0 software processes a top-level design (F), a control logic (C), and a peripheral component (D) to generate a bitstream, which is downloaded onto a target FPGA (G).

It is worth integrating a microprocessor core into the FPGA, because actual control systems include many parts that are not necessarily implemented in a logic circuit, e.g., user interface, network communication, etc. In such a case, most soft real-time tasks would be handled by a microprocessor, while hard real-time tasks are translated into a custom circuit, which might be attached as a peripheral device of a microprocessor.

VI. CONCLUSION

This study outlined a converter that translates PLC instruction sequence into logic description, with a design framework that integrates control logic and peripheral functions on an FPGA chip. Two sample ladder programs were examined and evaluated for Mitsubishi FX2N PLC and Altera APEX20KE FPGA. Various logic designs of these ladder programs were investigated and shown to fit into an off-the-shelf FPGA chip. The performance advantage over PLC technology was

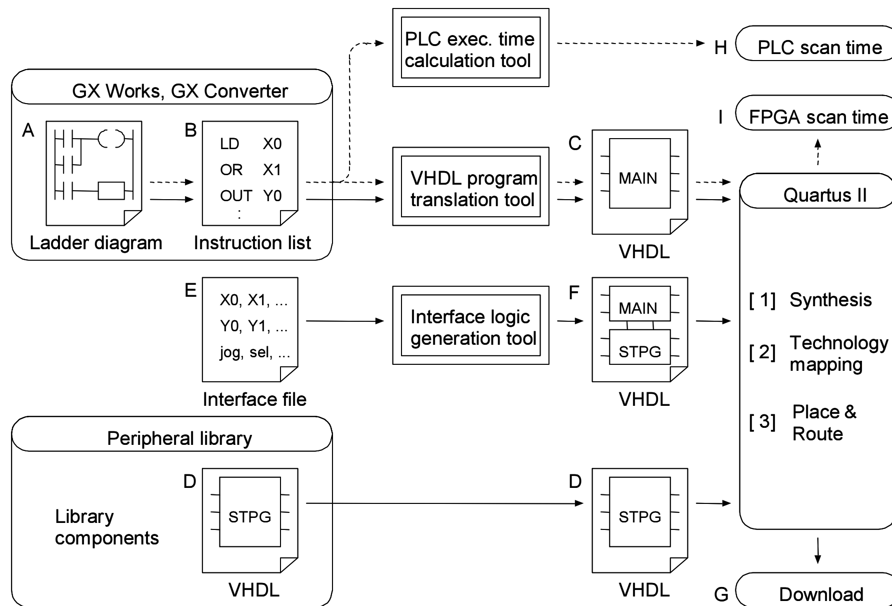


Fig. 5. Design framework.

obvious. In case of a productive ladder program, Sequential design was estimated to be 79 times faster than PLC, and Flat design was 43 times faster than Sequential design (i.e., 3397 times faster than PLC).

The authors are currently developing a "perfect layer winder" in conjunction with Yashima Netsugaku Co., Ltd., which is a specialty manufacturer of various fiber winders. To examine the feasibility of an FPGA control circuit in an actual control application, we have implemented a simple experimental winder using an Altera APEX20KE development board. This experimental winder operated successfully, driving a stepping motor and a linear slider with no problems. We are currently constructing a new winder with practical specifications, and will test our system with the new winder.

The following items are left for future study: (1) examination of more examples of control programs; (2) translation tool enhancement to support more control functions; (3) more performance optimization; and (4) investigation of system integration with embedded processors.

ACKNOWLEDGMENTS

The authors are grateful to Mr. Katsumi Asakura, the president of Yashima Netsugaku Co., Ltd. This work was partially supported by the Cooperation of Innovative Technology and Advanced Research in Evolutional Area (CITY AREA) and the 21st Century COE Program "Intelligent Human Sensing" from the Ministry of Education, Culture, Sports, Science and Technology of Japan. Support for this work was also provided by a Grant-in-Aid for Scientific Research from the Japan Society for the Promotion of Science (JSPS).

REFERENCES

[1] *IEEE standard VHDL language reference manual*, 2002, IEEE Std 1076-2002.
[2] M. A. Adamski and J. L. Monteiro, "PLD implementation of logic controllers," in *Proc. IEEE Int'l Symp. Industrial Electronics (ISIE'95)*, vol. 2, 1995, pp. 706–711.
[3] M. Adamski and J. L. Monteiro, "From interpreted Petri net specification to reprogrammable logic controller design," in *Proc. IEEE Int'l Symp. Industrial Electronics (ISIE 2000)*, vol. 1, 2000, pp. 13–19.
[4] M. Wegrzyn, M. A. Adamski, and J. L. Monteiro, "The application of reconfigurable logic to controller design," *Control Engineering Practice*, vol. 6, pp. 879–887, 1998.
[5] A. Wegrzyn and M. Wegrzyn, "Petri net-based specification, analysis and synthesis of logic controllers," in *Proc. IEEE Int'l Symp. Industrial Electronics (ISIE 2000)*, vol. 1, 2000, pp. 20–26.
[6] M. Ikeshita, Y. Takeda, H. Murakoshi, N. Funakubo, and I. Miyazawa, "An application of FPGA to high-speed programmable controller – development of the conversion program from SFC to Verilog –," in *Proc. 7th IEEE Int'l Conf. Emerging Technologies and Factory Automation (ETFA'99)*, vol. 2, 1999, pp. 1386–1390.
[7] I. Miyazawa, T. Nagao, M. Fukagawa, Y. Ito, T. Mizuya, and T. Sekiguchi, "Implementation of ladder diagram for programmable controller using FPGA," in *Proc. 7th IEEE Int'l Conf. Emerging Technologies and Factory Automation (ETFA'99)*, vol. 2, 1999, pp. 1381–1385.
[8] J. T. Welch and J. Carletta, "A direct mapping FPGA architecture for industrial process control applications," in *Proc. Int'l Conf. Computer Design (ICCD2000)*, 2000, pp. 595–598.
[9] Mitsubishi Electric Corp., *Programming Manual II: The FX series of programmable controller (FX1S/FX1N/FX2N/FX1NC/FX2NC)*, April 2003, JY992D88101 rev. D.
[10] ——, "MELFANS web," 2004, http://wwwf2.mitsubishielectric.co.jp/melfansweb/english/index.html.
[11] M. Chiang and R. Palkovic, "LCC simulators speed development of synchronous hardware," *Computer Design*, vol. 25, no. 5, pp. 87–92, 1986.
[12] E. Coffman, Ed., *Computer and Job-shop Scheduling Theory*. John Wiley & Sons, 1976.
[13] M. J. Gonzalez, "Deterministic processor scheduling," *ACM Computing Surveys*, vol. 9, no. 3, pp. 173–204, 1977.
[14] Altera Corp., *APEX 20K Programmable Logic Device Family Data Sheet*, March 2004, DS-APEX20K-5.1.