Control Eng. Practice, Vol. 1, No. 4, pp. 663-670, 1993
Printed in Great Britain. All rights reserved.

0967-0661/93 $6.00 + 0.00
© 1993 Pergamon Press Ltd

IMPLEMENTATION OF A PARALLEL ALGORITHM FOR EVENT DRIVEN PROGRAMMABLE CONTROLLERS

J. Park, N. Chang, G.S. Rho and W.H. Kwon

Seoul National University, ERC for Advanced Control and Instrumentation, Seoul 151-742, Korea

Abstract. In this paper, an event-driven programmable controller(PC) based on the dataflow logic solving unit is proposed. The proposed event-driven architecture is designed to solve the sequence logic programs while preserving the event order, which is very difficult in existing control-driven PCs. With the proposed parallel algorithm and the run-time optimization algorithm, the proposed PC eliminates brute force operations which are found in the conventional PCs. Using these algorithms, the proposed PC may solve about 1.33 instructions per event. The computer simulation shows that the proposed system with six solving units can solve one thousand instructions in 65 μsec in the worst case, when operating at 10 MHz. And in this paper, the hardware architecture for the above algorithm is suggested and the prototype system which was implemented using Xilinks FPGAs is also described.

Key Words. Computer control; control engineering applications of computers; digital control; industrial control; parallel processing; programmable controllers; sequence control

1. INTRODUCTION

A *programmable controller* (PC) is widely used in factories. Its applications can be found in many areas, such as chemical industry, manufacturing, and mining. The main purpose of the PC is to replace hard wired electro-mechanical relays used for sequence controls in factories. However, by using PCs instead of relays, there can be some problems.

The first problem is the logic program solving speed. When controlled by relays, there is only a small delay caused by mechanical relays. However, by using PCs, there can be lots of delay compared to the existing relay-controlled system. And this large delay of a PC may cause problems in sequence control loops. The overall processing time of the PC consists of logic-program-solving-time and I/O-update-time. I/O-update-time can be reduced by using a high speed I/O processor based on the multi-processor structure. By using a separate I/O processor, the I/O operation can be processed concurrently with the logic program solving. If the I/O processor and the communication between the main controller and the I/O processor are fast enough, the I/O-update-time can be ignored. In this case the logic-program-solving-time dominates the overall processing time of a PC. The logic-program-solving-time is the pure computation time in the PC. Since the basic data type of logic programs in a PC is a bit data type rather than a byte or word data type used in ordinary computers, it is not efficient to solve logic programs of a PC using ordinary microprocessors or computers. This means some dedicated architecture to manipulate bit data efficiently is required to improve the solving speed.

In recent years, several dedicated architectures for large-scale PCs have been proposed, and some of them are used in commercial programmable controllers. Many outstanding large-scale PCs are based on the multi-processor architecture or array processor architecture (Popovic *et al.*, 1990;Warnock, 1998; Shimokawa *et al.*, 1991). The logic solving processor(LSP) which was proposed by Kim *et al.*(1988, 1992) is an example of an array processor architecture. Their proposed LSP can execute eight logic instructions in every other clock cycle. Hence, the LSP can execute one thousand logic instructions in 22 μsec using eight cells when operating at 16 MHz clock rate. And it can execute one thousand instructions in 180 μsec if 25% of the block instructions are mixed in.

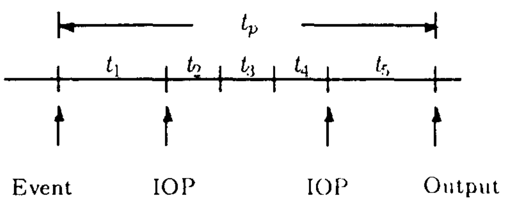

Figure 1: Time delay of PC

The second problem of a PC is slow speed due to the brute force solving. Most existing commercial PCs are control driven systems. In this kind of system, all of the logic programs are solved in each I/O scan interval regardless of the state of input points. However, in real plants, most input points are unchanged in a small time interval. Figure 1 shows a propagation delay, t_p, from an event occurrence to PC output. The propagation delay, t_p, can be divided into several terms, and among them, t_3 is the time to execute the instructions corresponding to the occurred event. Hence, the existing PCs using the brute force algorithm waste time, t_2 and t_4, in executing instructions which are not related to the occurred event in every scan interval.

The third problem is the occurrence of the undesired output due to the ignorance of the order of event. In real plants, all of the events do not occur simultaneously, but in existing PCs, input points are sampled periodically by a central processor and all of the events which occurs in one I/O sampling interval are treated as if they occurred simultaneously. As a result, in some logic programs, the later event can be treated before the former event. This situation can cause an undesired output and thus large time delay from the occurrence of an event to the normal response.

To eliminate the second and third problems, the PC itself should be operated by events. This event-driven PC has been studied in recent years. A Petri-Net based PC is an example of an event driven PC. Murata proposed a model for a PC based on the Petri-Net and implemented using an ordinary microprocessor (Murata *et al.*, 1986). Since their implemented system uses an ordinary microprocessor, the program-solving-speed is not as fast as a dedicated logic solving processor. Murakoshi proposed a memory based Petri-Net PC (Murakoshi *et al.*, 1990; Oumi *et al.*, 1991). The proposed model by Murakoshi includes a dedicated hardware logic solving part based on memory tables. While it is an event-driven sequence control system, the proposed system is hard to implement for a large-scale PC because it needs a large size of memory. The order of memory requirement is order of (*instructions* × (*input*+ *output*)). A dataflow logic solving processor is proposed based on the data driven concepts (Park *et al.*, 1991). In this system, however, only the logic solving unit is designed based on the dataflow architecture, so the overall system is not an event-driven system.

In this paper, to improve the logic solving speed, a dedicated logic solving unit instead of a general purpose microprocessor is proposed. And in order to reduce the memory size compared to the existing Petri-Net based PCs, a binary-dataflow-graph based PC is proposed. The operational strategy, parallel algorithm, and run-time optimization algorithm of the proposed PC are also proposed in order to increase the solving speed. And a hardware structure to implement the proposed algorithm is suggested, and its performance is estimated. Finally, a prototype system which is implemented using a FPGA is demonstrated.

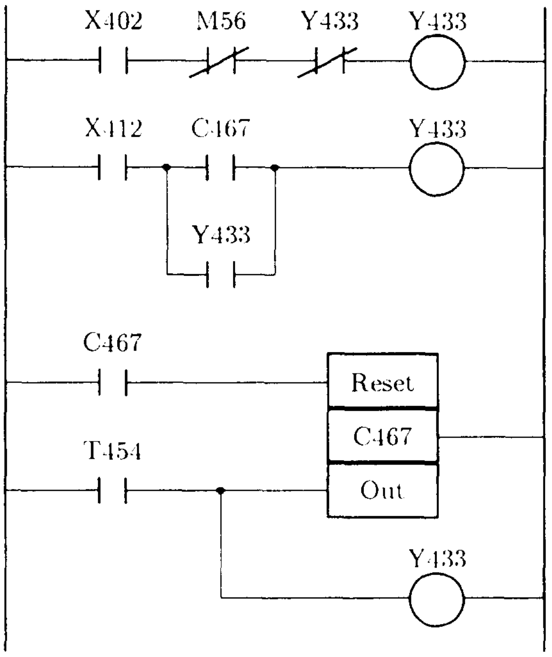

Figure 2: An example of a ladder program

2. LOGIC PROGRAMMING AND FORMALIZATION

To describe the architecture and operational strategy of the proposed PC in detail, a formalized representation of a sequential logic program is defined, and the basic operations of sequential control are described in this section.

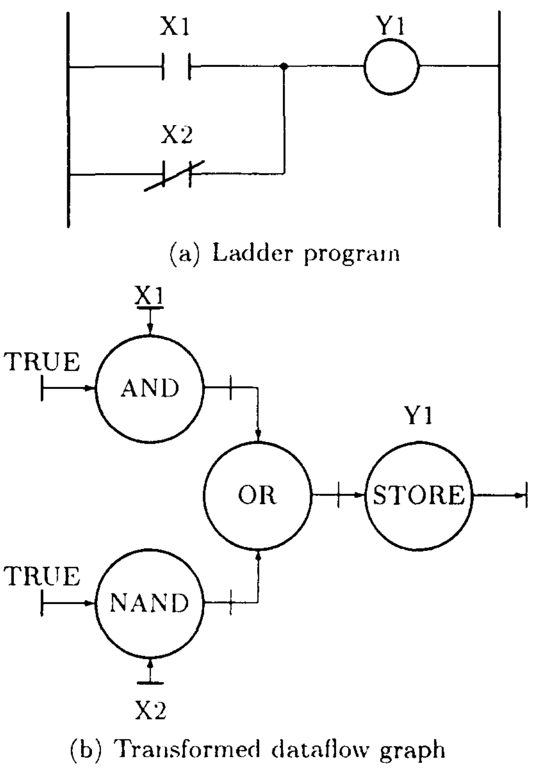

Figure 3: Ladder program and dataflow graph

Since a programmable controller (PC) was originally developed as a sequential control device, the language for the PC should be suitable for a sequential control problem. A sequence program is mainly operated asynchronously to the external events and concurrently with keeping data synchronizations. The conventional languages are difficult to use to describe a sequence control program which has the above properties. Currently, several kinds of programming language for PCs are used. Among them, a ladder language is one of the most widely used. The typical ladder language is shown in Fig. 2. The ladder language consists of the logic instructions and the block instructions. Most of the instructions used in the sequence control program are the logic instructions and the portion of the block instructions is below 25 percent. Though the ladder language is suitable to describe a sequence control, it is not easy to understand a large-scale sequence control program which is described in a ladder language. To resolve this difficulty, some other graphical modeling languages are experimentally used to model a sequence control. The Petri-Net, GRAFCET, and Mark Flow Graph are good examples (Murata *et al.*, 1986). These graphical languages, however, are difficult to solve directly by a PC hardware. In this paper, a sequence control program is modeled by a *Dataflow Graph (DG)* in a fine grain level. This dataflow graph model is suitable to describe an asynchronously operated sequence control program. The sequence control programs written in the ladder language can be transformed to the dataflow graph (Park *et al.*, 1991). In Fig. 3, a typical sequence control program written in the ladder diagram and the transformed dataflow graph are shown.

2.1. Formalization

The general dataflow graph can be formalized by Kavi(Kavi *et al.*, 1986). His formalized model is useful for the general dataflow computers. With some modification, his formalized model can be adopted for modeling a sequence control program. A dataflow graph for a PC is defined as

$$\begin{aligned}
\mathcal{G} &= (\mathcal{A} \cup \mathcal{L}, \mathcal{E}) \\
&\textit{where,} \\
\mathcal{A} &= \{a_1, a_2, \ldots, a_n\} \\
&\quad \textit{is a set of instructions,} \\
\mathcal{L} &= \{l_1, l_2, \ldots, l_n\} \quad \textit{is a set of links,} \\
\mathcal{E} &\subseteq (\mathcal{A} \times \mathcal{L}) \cup (\mathcal{L} \times \mathcal{A}) \\
&\quad \textit{is a set of edges.}
\end{aligned}$$

The instructions, a, in this definition, include logic instructions, block instructions and flow control instructions like fork and merge. The links, l, are treated as place holders of data values (tokens) which flow from one instruction to the next instruction. In a PC, the links are classified into three classes: input links, output links, and data links. The input links are used for modeling a sensor input in sequence control, the output links are used for modeling a relay output, and data links are used for modeling an intermediate result storage. The power nodes and input sensor nodes in a sequence control program form a starting link set $\mathcal{S}$ and the ground nodes form a terminating link set $\mathcal{T}$;

$$\begin{aligned}
\mathcal{S} &= \{l \in \mathcal{L} \mid (a, l) \notin \mathcal{E}, \forall a \in \mathcal{A}\}, \\
\mathcal{T} &= \{l \in \mathcal{L} \mid (l, a) \notin \mathcal{E}, \forall a \in \mathcal{A}\}.
\end{aligned}$$

For each instruction of a sequence control program, an input link set and an output link set can be thought of as

$$\begin{aligned}
I(a) &= \{l \in \mathcal{L} \mid (l, a) \in \mathcal{E}\}, \\
O(a) &= \{l \in \mathcal{L} \mid (a, l) \in \mathcal{E}\}.
\end{aligned}$$

Similarly, $I(l)$ and $O(l)$ for links can be defined. In PC programs, $|I(a)| > 1$ and $|O(a)| \geq 1$. In a dataflow graph, to represent the existence of a token in l, a marking M is defined as a mapping $M : \mathcal{L} — \{0, 1\}$. A link l is said to contain a token if $M(l) = 1$. In the proposed event-driven PC, a token in l represents an event in that link. A new marking, M', resulting from the executing of an instruction a at a marking M is denoted by $M \xrightarrow{a} M'$. An initial marking M_0 is a marking at the beginning of the solving of a logic program and a terminating marking M_t is a marking after the completion of solving a logic program. Solving a sequence control program can be thought as a mapping from the initial marking M_0 to the terminating marking M_t. We can think of an initially marked link set, $\mathcal{I}$, which satisfies $M_0 : \mathcal{I} \rightarrow 1$. $\mathcal{I}$ is a set of links which have a token at the beginning of the solving of a logic program. In ordinary PC logic programs, all of the starting links have tokens at the beginning, hence, $\mathcal{I} = \mathcal{S}$. After all sequences are solved, $M_t : \mathcal{T} \rightarrow 1$.

2.2 Operation Rule

The logic program solving in the PC can be thought of as a sequence of instruction execution. By the data driven concept, in a dataflow graph of a PC, an instruction a can be executed at a marking M if the following conditions hold;

$$\begin{cases} M(l) = 1 & for\ all\ l \in I(a) \\ M(l) = 0 & for\ all\ l \in O(a). \end{cases}$$

A new marking M' resulting from executing an instruction a at a marking M can be derived as

$$M'(l) = \begin{cases} 0 & if\ l \in I(a) \\ 1 & if\ l \in O(a) \\ M(l) & otherwise. \end{cases}$$

These executed instructions construct a solving sequence, σ. A marking M' is derived from M by a solving sequence σ and denoted by $M \xrightarrow{\sigma} M'$. The operation of the PC can be modeled by the set of solving sequence, σ, which holds $M_0 \xrightarrow{\sigma} M_t$.

3. OPERATION OF THE EVENT-DRIVEN PC

In this section, a model of an event-driven PC is proposed and the operations of the proposed system are described using the notations defined in the previous section.

3.1 Proposed Model

The basic assumption of an event-driven PC is that all of the events occur sequentially and none of them occur simultaneously. In real plants, this assumption is reasonable without loss of generality.

The dataflow graph representing the sequence control program in a PC, $\mathcal{G}$, can be split into many sub-graphs, $\{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n\}$. In the suggested event-driven PC, when an event occurs, the input/output processor generates a data token to the corresponding input-point-link, l_c. From l_c, a sub dataflow graph, $\mathcal{G}_c$, can be rebuilt as

$$\begin{aligned} \mathcal{G}_c &= (\mathcal{A}_c \cup \mathcal{L}_c, \mathcal{E}_c) \\ where, & \\ \mathcal{A}_c &= \{a \in \mathcal{A} \mid l_c \in I(a)^+\} \\ \mathcal{L}_c &\subseteq \mathcal{L} \\ \mathcal{E}_c &\subseteq (\mathcal{A}_c \times \mathcal{L}_c) \cup (\mathcal{L}_c \times \mathcal{A}_c) \\ I(a)^+ &= \{l \mid l \in I^{2n-1}(a), n = 1, 2, \ldots\} \\ I^n(a) &= \underbrace{I(I(I \cdots (a)))}_{n\ times}. \end{aligned}$$

In this case, $\mathcal{I}_c = I(O(l_c))^+ - O(O(l_c))^+$. The proposed event-driven PC solves $\mathcal{G}_c$ rather than $\mathcal{G}$ which is solved by existing PCs. In many cases, since the size of $\mathcal{G}_c$ is much smaller than $\mathcal{G}$, the logic solving time of the proposed PC is shorter than ordinary PCs. The executed instructions to solve $\mathcal{G}_c$ forms a solving sequence, σ_c.

All of the sequence control program can be transformed to a *binary dataflow graph* which has two input and two output edges, $|I(a)| = |O(a)| = 2$. Although the number of input links and output links are restricted, this binary dataflow graph can construct all of the sequence control programs (Murata *et al.*, 1986). In the proposed event-driven PC, the basic language is a binary dataflow graph. With this constraint, the hardware for the proposed model can be easily implemented because the binary model has a regular structure.

3.2. Parallel Algorithm

The solving sequence, σ, of the logic program can be split into sub-sequences $\sigma_1, \sigma_2, \ldots, \sigma_n$. Among these sub-sequences, two arbitrary sub-sequences, σ_i and σ_j can be solved concurrently by separate processing elements when

$$\{O(\sigma_i) \cap R(\sigma_j) = \emptyset\}\ and\ \{O(\sigma_j) \cap R(\sigma_i) = \emptyset\}$$

where

$$\begin{aligned} R(\sigma) &= I(\sigma) - O(\sigma) \\ I(\sigma) &= \{l \mid l \in I(a), a \in \sigma\} \\ O(\sigma) &= \{l \mid l \in O(a), a \in \sigma\}. \end{aligned}$$

In the proposed event-driven PC, when an event occurs, it is difficult to know which sub-sequence should be solved first. Furthermore, during the instruction execution, the data synchronization problem between the processing elements is important in a multiple processing element environment. The following algorithm is adopted to the proposed event-driven PC to clear the data synchronization problem. In the proposed algorithm, the matching-process is used to keep the data synchronization at a hardware level.

Parallel Algorithm

```
do while event $\notin \mathcal{T}$ {
    event ← queue
    if queue is empty then exit
    do while there is no idle LSU {
        if LSU #1 is idle then
            call LSU #1 algorithm
        elseif LSU #2 is idle then
            call LSU #2 algorithm
                    ⋮
        elseif LSU #n is idle then
            call LSU #n algorithm
    }
}
```

LSU #n Algorithm

```
do while $O(O(\text{event})) \notin \mathcal{T}$ {
    if fork instruction then {
        wait until fork register empty
        $O(O(\text{event}))$ → fork register
    }
    fork register → queue
    if $M(l) = 1$ {
        solve instruction
        event ← $O(O(\text{event}))$
    }
    else {
        wait until fork register empty
        $O(O(\text{event}))$ → fork register
        fork register → queue
        break
    }
}
```

3.3. Optimization Algorithm

An output can be unchanged even if some of the input points are changed. For example, if an output, Y, is $Y = X_1 \cdot X_2 \cdot X_3$, the changes of X_2 and X_3 do not affect the output when X_1 is 0. Using this property of a PC program, the run-time optimization algorithms for the proposed PC is summarized below.

Rule 1 The result of each instruction is compared to the previous data value. If they are same, stop running, otherwise, continue to the next instruction. In the first case, the terminating link set $\mathcal{T}$ is adjusted to $O(a)$.

Rule 2 At the merge point by an OR instruction, if one of the result is true, the matching process always succeeds.

With the application of these optimization rules to the solving sequence σ_c, a new minimal sub-sequence, σ_m, can be formed. The length of this sub-sequence, $|\sigma_m|$, satisfies $|\sigma_m| \leq |\sigma_c|$.

4. ARCHITECTURE OF AN EVENT-DRIVEN PC

In this section, a hardware architecture of the event-driven PC which is suggested in the previous sections is described. The hardware structure of the proposed PC consists of three main parts: the event driven IOP, the event queue, and the logic solving unit(LSU). The block diagram of the overall system is shown in Fig. 4.

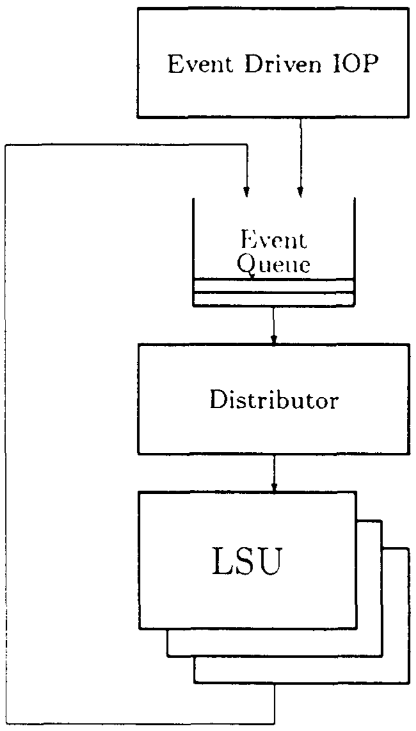

Figure 4: Block diagram of event-driven PC

4.1. Event-driven IOP

To make an event-driven PC, it is necessary to detect an event with a very high resolution of sampling interval. The event-driven IOP scans all the input points of the PC and checks whether any event occurs during the sampling interval. If an event occurs, it generates a data token and stores it in the event queue. To implement a high resolution event-driven IOP, it is necessary to make a flash comparator with a very large input width. The flash comparator produces an event when an input changes. However, in a large scale PC, since the number of input points is very large, it is difficult to make flash comparators. Hence, to implement an event-driven IOP for a large scale PC, distributed multiple IOPs which can handle an appropriate number of input points are needed.

4.2. Event Queue

In the proposed PC, the event tokens generated by the IOP and LSU are stored in the event queue. Although the IOP generates only one event token, the generated event token can generate multiple tokens through fork instructions during the solving process. In the proposed PC, since only as many tokens as LSUs can be enabled at the same time, the rest of the tokens are stored in the event queue. The tokens in the event queue are distributed to an idle LSU by a distributor.

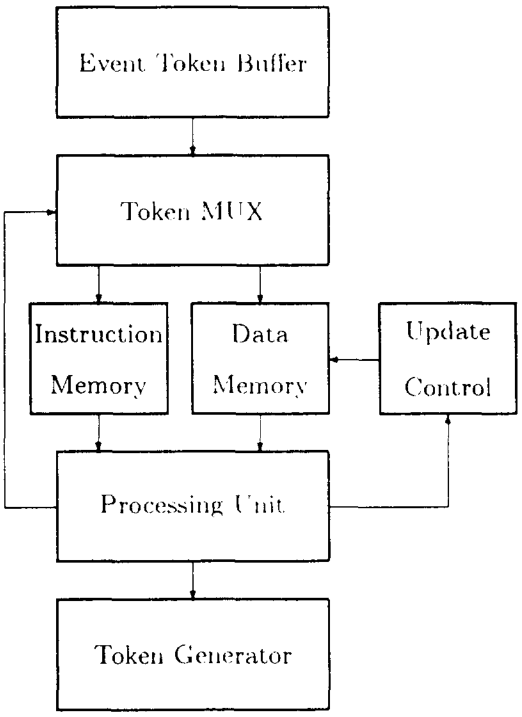

Figure 5: Block diagram of LSU

4.3. Logic Solving Unit(LSU)

The LSU consists of six sub-parts: a token mux, an instruction memory, a data memory, a processing unit, a token generator, and an update control unit as shown in Fig. 5. A token from the distributor selects the corresponding instruction, a, from the instruction memory. The selected instruction, a, is checked in the matching unit to see whether it can be solved immediately or not. If $M(l) = 1, \forall l \in I(a)$, the instruction a can be solved in the processing unit, or if one of the input links does not have a token, the event token which points out the unexecuted instruction is sent to the event queue for later processing. The executed result of an instruction in a processing unit is compared to the previous result for the run-time optimization. If the result is different, a new token is generated in order to execute the next instructions, $O(O(a))$. If the result is the same, no token for the subsequent instruction is generated.

5. PERFORMANCE EVALUATION

As mentioned in the previous sections, in the event-driven PC, there are three kinds of solving sequences when an event occurs; σ, σ_c, and σ_m. In the proposed event-driven PC, an optimized solving sequence σ_m is used. In this section, the performance of the proposed event-driven PC is analyzed. The performance of the suggested PC is measured by a computer simulation.

5.1. Conventional PC

In the existing PCs, all of the logic instructions are executed periodically regardless of the input point changes. And in this case, the program solving time is constant. The number of executed instructions, N_n, is $|\sigma|$.

5.2. Event-driven PC

In a logic program, it can be assumed that the input points are used uniformly. In this case, without the run-time optimization, the number of executed instructions, N_c, satisfies $N_c = Exp(|\sigma_c|) = |\sigma|/2$. If there is only one event in every IO scan time of the existing PCs, the suggested machine is twice as fast as the existing PCs. But, in a large scale PC, there can be more than two events in an IO scan period. In that case, the suggested machine is slower than existing PCs. Hence, a system with a large number of input points has a worse performance than existing control-driven PCs. However, in spite of the slower solving time, the proposed PC retains the order of events which are often neglected in control-driven PCs.

In the previous case, for each event, all of the instructions in σ_c should be executed. But with the run-time optimization algorithm proposed in the previous section, only a minimal sub-sequence, σ_m should be solved. From the run-time optimization rule, the expected number of instructions to be executed per event, N_m, can be obtained by

$$\begin{aligned} N_m &= Exp(|\sigma_m|) \\ &= (1-p) + p(1-p)\cdot 2 + p^2(1-p)\cdot 3 + \cdot \end{aligned}$$

$$= \lim_{n \to \infty} \sum_{k=1}^{n} p^{k-1}(1-p)k.$$

In the logic program, the probability that the result is changed from the previous result after solving a logic instruction, p, is 0.25. With this probability, the expected number of instructions, N_m, can be calculated as 1.33. This means that only 1.33 instructions per event may be executed by the proposed PC.

5.3. Computer Simulation

The performance of the proposed PC was evaluated by a computer simulation with respect to two parameters, the number of processing elements and the portion of stored instructions. Figure 6 shows the simulated processing time of the proposed event-driven PC with respect to the number of LSUs. From the simulation results, the proposed PC executes one thousand instructions in 210 μsec with one LSU, and in 65 μsec with six LSUs when they are operated at 10 MHz. From the computer simulation results, it can be said that the percentage of the stored instructions hardly influences the program solving time. From the simulation results and the calculated values of N_m, it takes 279 μsec with one LSU and 100 μsec with four LSUs to execute the corresponding instructions when an event occurs. This solving time may be nearly constant regardless of the logic program length.

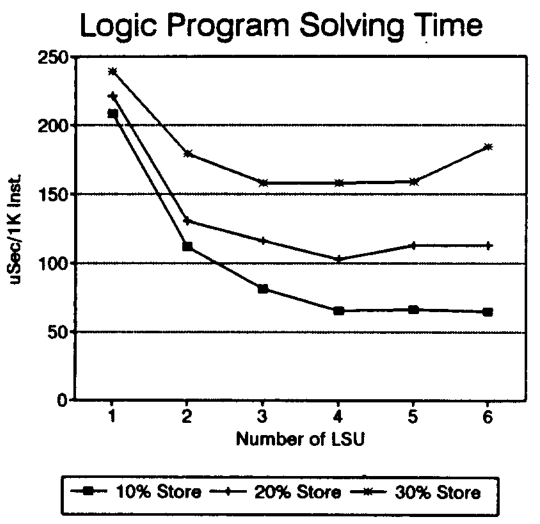

Figure 6: Computer simulation result

6. IMPLEMENTATION OF PROTOTYPE SYSTEM

The prototype system of the proposed PC is made using FPGA and a microprocessor (see Fig. 7). Although the event-driven IOP can be constructed by a flash comparator, it is not efficient to handle a large number of I/O points. In the prototype system, the event-driven IOP is implemented using the Intel 80186 microprocessor. This IOP scans all of the input points periodically with a short interval to emulate the event-driven IOP. If it detects an input change, it generates an event token and stores the generated token into the event queue. The IOP keeps on scanning the input points without waiting the completion of the LSU operation. The event queue is implemented using a two port FIFO (first in first out) memory. The tested prototype system has four LSUs which are implemented using Xilinks PGAs. Since the Xilinks XC3000 series PGA cannot contain RAM inside, the instruction and data memory are located outside the LSU. Due to the usage of the external RAM in the prototype system, the operating speed of the proposed PC was limited to 10 MHz clock speed. If the internal RAM or a high speed RAM is available, the operating speed can be increased.

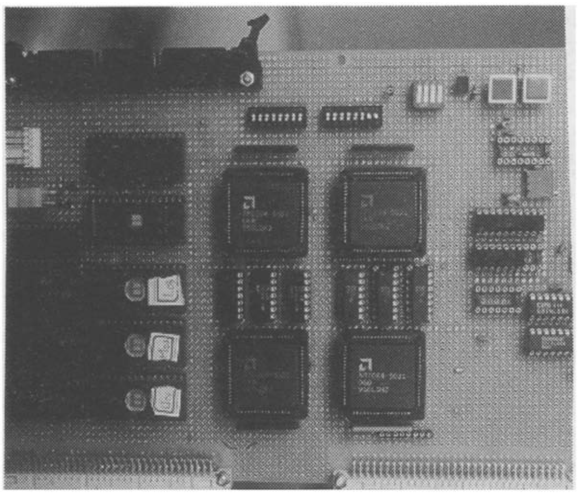

Figure 7: Photograph of prototype system

7. CONCLUSION

An architecture of an event-driven PC is proposed in this paper. The proposed PC is designed based on an event driven architecture in order to implement the real sense of sequence control which is poorly implemented by existing PCs. To implement the event driven concept at the hardware level, the logic solving unit of the proposed system is designed based on a dataflow machine which has concurrent processing and data synchronization capabilities at a fine grain level. As the proposed system uses a parallel algorithm and run-time optimal algorithms, the performance of the proposed system is very high. With the proposed optimization rule, the average number of instructions to be executed per event is only 1.33. This result shows that the performance of the proposed system can be better in speed than other existing PCs while keeping the event-order in solving the sequence program. The computer simulations and the implemented prototype system show that the logic solving time of the proposed hardware is 210 μsec with one LSU and 75 μsec with four LSUs when operating at 10 MHz.

While the proposed PC shows a good performance while retaining the event order, there are several things to be improved. First, the architecture of the PC needs to be extended to include block instructions which are more complex and take a longer time to solve. Second, the property of the binary decision structure in the proposed system needs to be analyzed. Third, an off-line optimization algorithm in addition to the runtime optimization one can be proposed for higher performance, by utilizing the special properties of logic programs in PCs.

8. REFERENCE

Kavi, K.M., B.P. Buckles, and U.N.Bhat(1986). A Formal Definition of Data Flow Graph Models, *IEEE Trans. on Computers*, Vol. 35.

Kim, J., W.H. Kwon, and J. Park(1988). Architecture of a logic solving processor for programmable controller. *Proceedings of '88 SICE.*

Kim, J., J. Park, and W.H. Kwon(1992). Architecture of a ladder solving processor(LSP) for programmable controllers. *Microprocessor and Microsystems.*, Vol. 16, No.7.

Murata, T., N. Kormoda, K. Matsumoto, and K. Haruna(1986). A Petri net based controller for flexible and maintainable sequence control and its applications in factory automation, *IEEE Trans. Industrial Electronics*, Feb.

Murakoshi, H. and Y. Dohi(1990). Petri net based high speed programmable controller by ASIC memory, *Proceedings of '90 SICE.*

Oumi, T., G. Ding, H. Murakoshi, M. Sugiyama, Y. Dohi, Y. Gai, H.M. Shih(1991). A Petri net generating development support system, *Proceedings of '91 IECON.*

Park, J., N. Chang, and W.H. Kwon(1991). An architecture of dataflow LSP for programmable controllers, *Preprint of '91 IFAC Workshop on AARTC.*

Popovic, D., and V. Bhatkar(1990). *Distributed computer control for industrial automation*, pp.566-576, Marcel Dekker, Inc.

Shimokawa, Y., T. Matsushita, H. Furuno, and Y. Shimanuki(1991). A high-performance VLSI chip of programmable controller and its language for instrumentation and electric control, *Proceedings of '91 IECON.*

Warnock, I(1988). *Programmable controllers - operation and application*, Prentice Hall.