Implementation of a RISC microprocessor for programmable logic controllers

Gab Seon Rho, Kyeong-hoon Koo*, Naehyuck Chang*, Jaehyun Park*, Yeong-gi Kim[†] and Wook Hyun Kwon*

A special purpose RISC (reduced instruction set computer) microprocessor for programmable logic controllers (PLC), named PLCRISC, is proposed. To develop an optimal PLCRISC, we analysed existing PLC programs currently used in factories, with special attention to the instruction execution characteristics and features required for a high performance PLC processor. Based on this analysis, an optimal RISC-style instruction set and an architecture suitable for the required features are suggested. In particular, the instruction format, the instruction pipeline, and the detailed internal architecture are the significant characteristics of the proposed PLCRISC. The performance enhancement achieved with a PLCRISC is seen from a straightforward evaluation. ASIC implementation with VHDL is also discussed. The PLCRISC is under fabrication in a 0.8 μm CMOS technology.

Keywords: programmable logic controllers, RISC architecture, special purpose microprocessor

The programmable logic controller (PLC) was originally developed as a sequential control device to replace electro-mechanical relays in factories, and it is now being used for numerous applications, including in factory automation in chemical processing, manufacturing, and mining[1]. With the progress of micro-computing technology, the PLC is improving at a very fast pace. Factories are also expanding and becoming more complex at a very fast pace. Ideally, the PLC will improve in step with factory expansion. It must provide more computing power and functions.

In many industries, a commercial general purpose microprocessor is used as the central processing unit of the PLCs. Indeed, many powerful microprocessors provide high performance in general purpose applications. However, such processors cannot guarantee the high performance required for PLCs for the following three reasons. First, PLCs operate best with bit type data structures, while general purpose microprocessors are designed to handle byte or word type data[2]. Therefore, if the PLC employs a general purpose microprocessor, a bit type operation must be made of several instructions. This results in significant performance degradation.

Second, for control purposes, the PLC receives large volumes of input data that change very frequently according to the states of a controlled system. Since the PLC must use these data as operands of its instructions instead of the internal data that are stored in the registers, the PLC processor must fetch the data from the memory for every instruction. Thus, the memory traffic in PLCs becomes very heavy. But general microprocessors do not adequately meet this high memory bandwidth, because they have an architecture suitable for manipulating data in internal registers. Therefore, an improvement in memory bandwidth is required to enhance the overall performance of PLCs.

Third, for high performance PLCs, an efficient interactive mechanism between bit type operation and word type operation must be available, which will be explained in detail in the following section. General purpose microprocessors do not have this interactive mechanism.

An approach to overcome the above problems is to use a special purpose processor. In recent years, several new architectures of the PLC processor have been proposed. One of them is to use a dedicated hardware logic which is based on the array processor architecture[2]. This processor can execute bit instructions very fast due to optimization of bit operations. However, it has difficulty in executing word instructions and also has an inflexible configuration. Another type of architecture is to use a memory-based implementation[3]. This processor can perform fast parallel computation, and its structure is very simple. However, it

*Department of Control and Instrumentation Engineering, Seoul National University, Shinlim-dong San56-1, Kwanak-gu, Seoul, 151-742, Korea Email: rho@cisl.snu.ac.kr
[†]Samsung Aerospace Industries, Ltd, Suwon, PO Box 111, Kyungki-do, 441-600, Korea
Paper received: 29 August 1994. Revised: 11 August 1995

requires large memory fields because its data structure is very sparse and many bits of the data are used ineffectively.

A more efficient method to make a high performance PLC is to develop a special purpose processor in order to meet application-specific requirements obtained through the analysis of existing PLC programs. This method is suitable especially for a special purpose processor that adopts a RISC architecture[4, 5]. Recently a RISC-based PLC processor was proposed by Toshiba[6], but Toshiba did not address the analysis of the PLC program characteristics. Moreover, Toshiba could not describe an instruction set suitable for high performance PLCs.

In this paper, a special purpose RISC microprocessor for PLCs, named PLCRISC, is proposed. The PLCRISC is developed to have a limited and simple instruction set, and it utilizes hardware resources optimally to speed up the most frequently used instructions through highly efficient VLSI design technology[4]. In the PLCRISC, a RISC-style instruction set is selected optimally through the analysis of many PLC programs currently used in factories. Thus, the PLCRISC supports more general instructions than can the above processors, and consequently it can provide a more flexible programming environment. Several function units that balance bit operations and word operations are implemented in order to enhance the overall processing speed.

In the next section, we characterize existing PLC programs in use for various factory automation. From this characterization, we can suggest the features required for high performance in PLC processors. In the following sections, the instruction set, the instruction format, the instruction pipeline, and the internal architecture of the PLCRISC are described. Lastly, performance evaluation and development in ASIC are discussed.

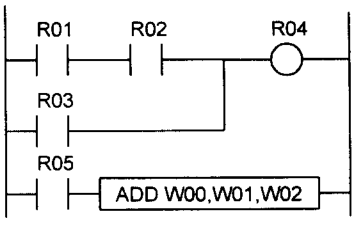
CHARACTERISTICS OF PLC PROCESSORS

In this section, the execution characteristics of PLC programs are described, and features required for a high performance PLC processor are suggested.

PLC programming

Many programming languages are available for PLCs, such as the instruction list (IL), the structured text (ST), the function block diagram (FBD), and the ladder diagram (LD)[7]. Among them, the LD language is the most popular, and it has a graphical form that is suitable for the representation of a sequential control system. The LD language is depicted by a diagram of matrix type symbols that represent relays, switches, solenoids and lamps, etc. This graphical language, however, is not easily executed directly by a PLC processor. Generally, the programs written in the LD language are converted to mnemonic PLC instructions for execution. *Figure 1* shows an LD language program and its corresponding mnemonic PLC instructions.

Although the mnemonic instructions are supported somewhat differently by each PLC manufacturer, the



```
STR    R01
AND    R02
OR     R03
OUT    R04
STR    R05
ADD    W01,W02,W03
```

(a) a ladder diagram program (b) converted PC instructions

Figure 1 An example of a ladder diagram program

typical PLC will usually support two primary groups of instructions: bit instructions and word instructions. In the bit instruction group, bitwise logic instructions, such as START, AND and STORE, are included. These instructions have a one-to-one correspondence to the relay logics that were used as sequential control devices in the past. Word instructions consist of compare, arithmetic, move, and flow control instructions[8]. The word instructions can be used in functions such as PID control, user interface, and communication.

The word instructions are very similar to the instructions of general purpose microprocessors. However, in PLCs, the word instructions can be tightly coupled with bit instructions, and thus they have different execution patterns compared to general purpose microprocessor instructions.

Frequency of PLC instructions

Because PLCs are used in a variety of applications, such as chemical processing, manufacturing, mining, pulp and metal processing, each unique PLC requires control programs suitable for its specific application. These control programs have different execution patterns. Furthermore, because the users of PLCs have different programming styles, the frequency of use of PLC instructions varies greatly among users.

To analyse the behaviour of PLC programs, we investigated many PLC programs currently used in factories. We simulated their execution on a computer and stored the results. In particular, we focused on several measurements: frequency of instructions, frequency of addressing mode, and execution sequencing[9]. All of the measurements are dynamic measurements that are obtained by counting the actual number of executions for each instruction. Such measurements can provide more useful information than static measurements, which merely count the source codes of the programs. Among these measurements, the frequency of PLC instructions, a key parameter, can be presented as a percentage of total instructions (see *Table 1*).

For efficient execution of the instructions used frequently in PLC programs, a special processor is needed. The frequently used instructions in PLC programs, such as bit instructions, move instructions and compare instructions, must be included in the machine instruction set of the special purpose processor to obtain a high performance PLC. *Table 1* provides useful information when designing the machine instructions of a PLC processor.

Table 1 Frequency of PLC instructions

Instruction type	Percentage
Bit instruction	69.0
Timer, counter	1.4
Move	12.0
Compare	9.3
Arithmetic	6.1
Flow control	2.2

Bit instructions of PLCs

Analysis of PLC instructions shown in *Table 1* shows that two-thirds are bit instructions. This occurs because the PLCs have been used as sequential controllers replacing the electro-mechanical relays. Because the states of relays can be represented by only two states, ON and OFF, the bit instructions use bit type data structures as their operands and perform bitwise logic operations. However, general purpose microprocessors have byte or word data structures, and thus they are fundamentally designed for types of instructions other than the bit instructions of the PLC programs[2].

Generally in commercial PLCs, the bit instructions of PLC programs are emulated by several instructions of the general purpose processor. This emulation produces a large software overhead, namely, much more software programming is provided than is required. Because the bit instruction is most frequently used in PLC programs, the overhead degrades the performance of PLCs greatly. Therefore an increase in the execution speed of bit instructions is one of the most important factors in improving the overall performance of a PLC processor.

Execution sequence

For complete execution of a PLC program, the PLC samples input signals from the controlled system, process the value of the signals according to the preprogrammed logic, and update the values of output signals. This procedure is called 'one scan'. The PLC repeats this scan cycle continuously.

Because the PLC replaced relay sequential logic where the relays operated sequentially, the PLC program tends to be executed straight from top to bottom without a loop or branch during one scan cycle. At least this is true for a program that consists of only bit instructions. *Table 2* shows the frequency of executions for each line of source code of the programs analysed. Almost all instructions are executed only once during one scan cycle. This is because few flow control instructions are used in PLC programs as shown in *Table 1*.

In general purpose microprocessors, most programs do not access all instruction codes uniformly. This rule is called the principle of locality[10]. Based on this hypothesis, the memory hierarchy using a cache memory can produce better performance. But in the PLC, there is no locality, as shown in *Table 2*. Therefore the implementation of a cache memory in a PLC processor is less efficient.

Table 2 Execution number of each line

Number of executions during one scan cycle	Percentage
Only once	98.1
Twice	1.8
More than twice	0.1

Memory bandwidth

In factories, the PLC is connected with sensors and actuators for receiving input data from the sensors and sending driving commands to the actuators. The PLC processor receives the input signals and stores them into the main memory. After the logic programs are processed with the given input data in the main memory, the output values are updated and sent to actuators and ultimately reflected in changes to the external system. During the execution of the PLC program, most of the PLC instructions require memory resident operands. This is especially true for the bit instructions that are frequently used in PLC programs and fetch their operands for every execution. Thus in PLCs, many memory references are needed. *Table 3* shows the occurrence of the types of operands, which was obtained by the examination of the dynamic behaviour of PLC programs.

The memory resident operand needs an operand fetch cycle (OFC) and results in a conflict with an instruction fetch cycle (IFC). Because most of the processors in commercial PLCs have a single external bus, the instruction and the operand must be fetched sequentially. *Figure 2* compares the sequential execution timing of a single bus pipeline architecture machine with the parallel execution timing of a separated bus pipeline architecture machine. As shown in *Figure 2*, if a processor has two separate buses for instruction and operand, the two cycles can occur in parallel. To meet the high memory bandwidth shown in *Table 3*, a processor that has two separated buses must be employed in the PLC.

Table 3 Frequency of operand sources

Operand type	Percentage
Memory reference	75.0
Immediate scalar	20.0
No operand	5.0

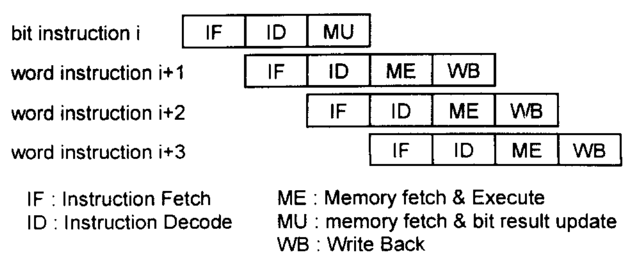


Figure 2 Sequential and parallel execution

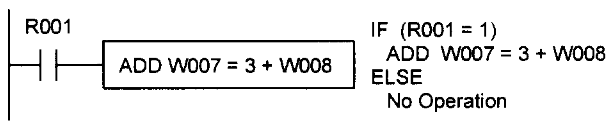


Figure 3 Execution of a word instruction

Interaction between bit and word instructions

In PLCs, the word instructions, such as addition, multiplication and shift, are related closely with the bit instructions. For example, the branch instruction checks not only the branch condition but also the value of a bit accumulator, which stores the resulting data of bit instructions. As another example, both memory updating and data writing to the registers are decided by the value of the bit accumulator. As explained above, the word instructions of PLCs are executed conditionally according to the result of a bit instruction. *Figure 3* shows an example of a word instruction in the LD language and its corresponding execution procedure.

Generally in commercial PLCs implemented with typical microprocessors, the conditional execution of a word instruction is performed in software using test and jump instructions. Since it takes too much time, the conditional execution of a word instruction produces a software overhead in typical PLCs.

For the efficient interaction between bit and word instructions, a bit processing unit must be included in the PLC processor, and special hardware logic for the fast decision making on the conditional execution of word instructions must be implemented. Furthermore, in a PLC, a bit instruction often needs an operand from the word instructions, for example, after a compare instruction is executed with word data, the following bit instruction takes the result as its operand. Thus the PLC processor must provide a method where the results of some word instructions can be used as operands of bit instructions.

INSTRUCTIONS OF THE PLCRISC

Using the analysis of PLC execution characteristics described above, we can design an optimal RISC-style instruction set, particularly ensuring that it efficiently accommodates the predominance of bit instructions in PLCs. Most currently used PLCs derive from microprocessors that have complex control logic and word instructions. The improvement that a PLCRISC offers is that it has a core of simple and limited instructions in a fixed format. This core instruction set can manage almost all of the required execution of PLC instructions.

During the design and implementation stage of the PLCRISC instruction set, the decision between what is to be implemented in hardware and what is to be done in software has been made. Some PLC instructions require many execution stages, different instruction formats, and various addressing modes. For example, one particular complicated multiplication instruction requires three memory references, two for operand fetches and one for a result writing. A processor that has such PLC instructions as its inherent instruction is very difficult to design, and its implementation costs are very high. And because such processors have very complex control logic, the bit instructions that carry out simple operations cannot be executed quickly and hardware resources cannot be utilized optimally. Therefore, for fast execution of PLC instructions, a simple and limited instruction set that has a fixed format is more useful.

Instruction set

The PLC instructions that are used frequently in the PLC programs, as shown in *Table 1*, were selected to be implemented in the PLCRISC. Since the bit instructions of PLC programs have simple semantics, all bit instructions are implemented as the instructions of the PLCRISC. Still, some complicated PLC word instructions must be executed in the PLCRISC, which has a RISC architecture. This is a difficult obstacle, but it can be overcome by breaking down the necessary PLC word instructions into several simpler instructions that the PLCRISC can manage efficiently.

In the PLCRISC, more than 80 instructions, such as big logic instructions, memory access instructions, data move instructions, arithmetic instructions, shift and rotate instructions, compare instructions, and flow control instructions, are implemented. Several representative instructions of the PLCRISC are shown in *Table 4*.

Table 4 Part of the PLCRISC instruction set

Opcode	Operands	Description
STR	$Bit_addr	Start rung
AND	$Bit_addr	And bit
OR	$Bit_addr	Or bit
INIOUT	$Bit_addr	Initializer out
OUT	$Bit_addr	Out bit
MCS		Master control set
MCR		Master control reset
LOAD.l(h)	Reg,#const	Load immediate
STORE.l(h)	$W_addr,Reg	Store absolute
MOVE	Src,Des	Move data between regs.
ADD	Src1,Src2,Des	Add
SUB	Src1,Src2,Des	Subtract
MUL	Src1,Src2,Des	Multiply
DIV	Src1,Src2,Des	Divide
RLC	Src1,Src2,Des	Rotate left with carry
SHL	Src1,Src2,Des	Logical shift left
BSET	Bit_pos,Des	Bit set
STRCEQ	Src1,Src2	Compare equal (TOS)
BIN	Src,Des	Convert BCD to binary
BCD	Src,Des	Convert binary to BCD
LOOP	Reg,#LABEL	Loop if reg is non zero
JMP	Reg	Jump indirectly
CALL	#LABEL	Subroutine call indirect
RETI		Return from interrupt
TRAP		Software trap interrupt

While almost all bit instructions of a PLC program have one-to-one correspondence to the bit instructions of the PLCRISC, bit output instructions and pulse instructions are substituted by two or three bit instructions, as they access memory twice or three times respectively. As in RISC processors used for other applications, only load and store instructions can access memory. Usually all bit instructions require absolute addressing operands. And the word instruction that needs a complex addressing mode is broken into several instructions that have a simple addressing mode. Thus absolute, immediate, and register indirect addressing modes are implemented in the PLCRISC.

Move instructions are supported to transfer data between internal registers. For word arithmetic operations, 12 arithmetic and logic instructions are defined, including addition, subtraction, multiplication, shifting, rotating, and binary-BCD conversion. In the PLCRISC, the result of a compare instruction is stored in the bit accumulator, and thus any word instruction can be conditional if it follows the compare instruction. Seven flow control instructions are included. The call instruction saves the return address at the special register. The branch instruction is not expected to be taken, and the branch condition is checked at an early stage to reduce the branch penalty.

In the PLCRISC, the floating point operations are performed using an external coprocessor. The choice of adding an external coprocessor is optimal because few floating point instructions are used in PLC applications. Possibly, a special application or intelligent control algorithm sometimes requires floating point operations. Thus, judging from the perspective of performance improvement *versus* hardware requirements, an implementation of a floating point unit in the PLCRISC is less efficient. Though the PLCRISC has no coprocessor instruction, it supports the interface with the floating point coprocessor in the form of a memory instruction. The information that is needed for a floating point operation is encoded in the memory access instruction and transferred to the external floating point coprocessor.

In a RISC processor, the branch mechanism has one of the largest impacts on the performance of the machine. The branch instruction breaks the pipelined streams, so minimization of their effects is important. In the PLCRISC, the branches are taken after checking both the result of a bit instruction and the corresponding branch condition, as mentioned earlier. To do so, compare instructions are separated from branch instructions, and the condition check is performed in the early pipeline stage of a branch operation.

Figure 4 represents the instruction formats of the PLCRISC. The size of all instructions is 32 bits. The addressing range of bit instructions is 16 Mbits, and the addressing range of word instructions is 16 Mbytes. For future definition of instructions, an instruction's operation code is divided into group code and instruction code.

A condition field is prepared to enable or disable the conditional execution of word instructions. Every word instruction has the condition field specifying the mode of execution, which is either unconditional or conditional. If the mode is conditional, the instruction is executed according to the value of the bit accumulator, which accumulates the result of the bit instructions. When a conditional word instruction is executed, the bit accumulator is tested before the instruction changes the context of the processor, such as the contents of the register file, an activation of memory access, and the value of the program counter. If the tested value is not one, the instruction cannot take effect on the context of the processor.

Instruction pipeline

In the design of the PLCRISC pipeline, the key concern is to speed up the bit instructions that are the most frequently used PLC instructions. All bit instructions require at least one memory operand access, and thus one execution cycle of a bit instruction consists of an instruction fetch cycle, a decoding cycle, an operand fetch cycle, a bit data computation cycle, and a result updating cycle. Since an internal cache is not implemented in the PLCRISC due to the reason mentioned in an earlier section, the two memory fetch cycles dominate the execution cycle of bit instructions. A way to minimize the impact of this dominance is to implement the bit data computation and result updating in the same pipeline stage with the operand fetch cycle. In the PLCRISC, the bit data computation and the result updating

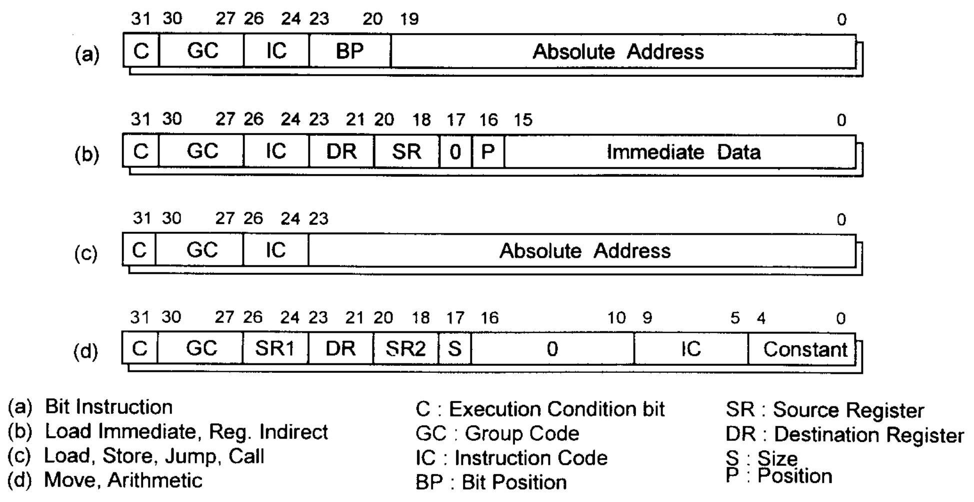


Figure 4 Instruction formats of the PLCRISC

can be performed quickly with a bit processing unit. And since the result of the bit instruction determines the conditional execution of the following word instruction, the result must be calculated before the word instruction takes effect on the context of the PLCRISC. Lastly, although the decoding time of the PLCRISC is very short, the decoding cycle is performed separately from the instruction fetch cycle in order that the exceptions or interrupts can be processed in this cycle. Therefore, a three-stage pipeline is suitable for efficient execution of bit instructions in the PLCRISC.

Because some word instructions need more complex and time-consuming operations, more pipeline stages are required. Generally word instructions require an instruction fetch cycle, a decoding cycle, an operand fetch cycle, an arithmetic or logic computation cycle, and a result write-back cycle. While more pipeline stages may increase the execution speed of word instructions, the pipeline control logic and the internal forwarding control logic nonetheless become complex. Furthermore, additional pipeline stages do not always increase the execution speed of bit instructions because bit instructions need only three pipeline stages. Thus, it is necessary to reduce the pipeline stages of word instructions.

In the PLCRISC, a word instruction that needs both an operand fetch cycle and an arithemtic computation cycle are broken into two separate instructions. Because some instructions need a lengthy arithmetic computation, the arithmetic computation cycle and the result write-back cycle are performed in independent pipeline stages.

As a result, a four-stage pipeline architecture for word instructions is adopted in the PLCRISC. As shown in *Figure 5,* four instructions are executed simultaneously in each pipeline stage. To balance the load of each pipeline stage, all of the instructions are analysed with regard to the time required by each operation. During the implementation of the instruction set, the PLCRISC instructions are redefined or modified to maximize the utilization of the pipelining.

In the instruction fetch stage, an instruction is fetched from the instruction bus, and the program counter is incremented. Because the instruction bus has a width of 32 bits, and because the maximum number of possible instructions is fixed, all instructions are fetched in one cycle. The instruction is decoded in the instruction decoding stage and hence produces the control signals for each pipeline stage. For a branch operation, branch conditions are checked and the branch target address is selected according to the result. Exception conditions, such as memory access miss, division by zero, and trap instruction, are also checked. At the same

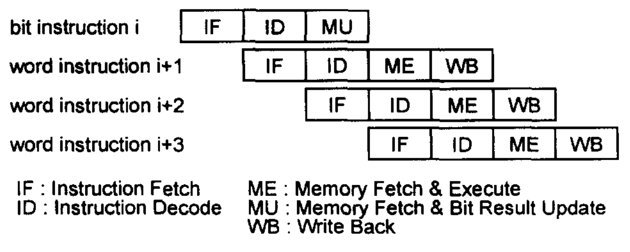


Figure 5 PLCRISC pipeline

time, operand source registers are selected and an absolute address is latched into the memory address register.

During the memory fetch and execution stage, a memory access occurs for load and store instructions. For bit instructions, an operand reading and a logic computation are performed. The valid bit result is provided at the end of this stage, and it can be used in the decoding stage of the next instruction. For word instructions, only an arithmetic computation is executed in an execution unit. In the write-back stage, the result of a word instruction is stored in the register file.

ARCHITECTURE OF THE PLCRISC

The key point of the PLCRISC architecture is the design philosophy of simplicity and efficiency. It has an architecture especially designed for single-cycle execution, simple load/store interface to memory, and simple fixed-format instructions. The architectural simplicity yields a reduction in chip area, and the saved chip area can be utilized by other hardware resources, such as multipliers, barrel shifters, and binary-BCD converters[11-13].

Basically, the architecture of the PLCRISC is designed to satisfy the features for a high performance PLC processor as mentioned earlier. For fast execution of bit instructions and simple word instructions, a pipelined architecture is adopted, and every instruction can be executed within one cycle. The separated instruction bus and operand data bus increase the memory bandwidth to provide parallel operation of the instruction and operand fetch. The bit processing unit supports fast bit data manipulation, and numerous functional units are provided for fast execution of word instructions. The conditional execution of a word instruction is performed in a hardware level to avoid performance degradation. The PLCRISC has four internal units: an instruction register unit (IRU), an execution unit (EU), a program counter unit (PCU), and a memory interface unit (MIU).

Instruction register unit

The IRU fetches an instruction for execution, which is referred by the program counter register in the PCU, and decodes it to produce appropriate control signals. *Figure 6* shows the block diagram of the IRU.

The instruction decoder is made by hardwired logic, which results in a smaller number of gates than microprogramming. Because the PLCRISC has four pipeline stages, the control signals are delayed by one cycle for the memory fetch and execution stage and two cycles for the write-back stage. The bypass logic takes care of bypassing the register file when a source register corresponds to a destination register for previous instructions that have not yet written back to the register file. The logic provides two levels of internal forwarding so as to avoid pipeline hazards.

The delay logic also performs the conditional execution of word instructions and provides the efficient interaction between bit and word instruction. When a word instruction

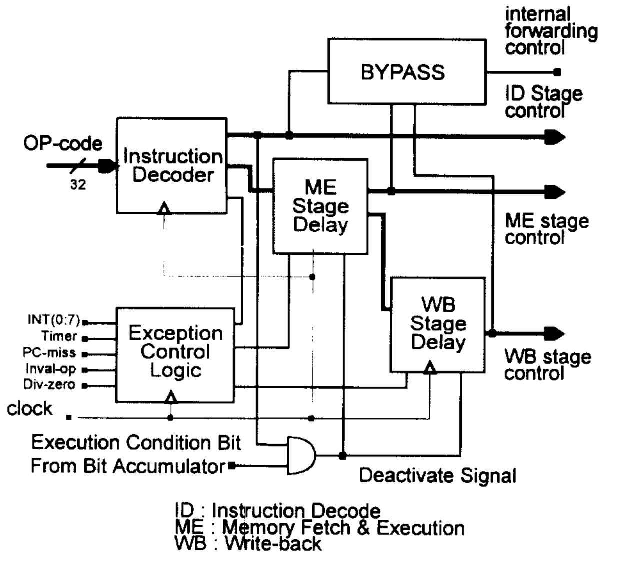


Figure 6 Block diagram of the instruction register unit

is in the mode of conditional execution and execution is disabled, the delay logic deactivates the control signals for the memory fetch and execution stage and the write-back stage. This logic decides the conditional execution of word instructions very quickly, and provides no difference in execution time between conditional execution and unconditional execution. Therefore, the user of the PLC can estimate the execution time of a program regardless of conditional execution. This advantage allows use of the PLCRISC in real time application, in which estimation of the program execution time is regarded as one of the most important factors.

The IRU is responsible, furthermore, for exception and interrupt handling, and the exception control logic contains a finite-stage machine for this purpose. The PLCRISC provides eight external interrupt lines, so it manages eight levels of interrupts. For the simplicity of external logic, the interrupts are received in the form of autovectors as in the MC68000 family of processors. Several internal exceptions, such as instruction access miss, data access miss, and invalid op-code, are supported in this unit. The internal timer interrupt is used in a periodical program. A trap instruction supported by this unit makes it possible to debug the PLCRISC program easily.

Execution unit

In the EU, several hardware resources essential to all logic and arithmetic operations are implemented. It consists of a 32 bit register file and function units, such as a bit processing unit, an arithmetic logic unit (ALU), a multiplier, a barrel shifter, a binary-BCD converter, a magnitude comparator, and a step divider.

This unit reads the operands from registers in the instruction decoding stage, processes logic and arithmetic operations in the memory fetch and execution stage, and writes the results into a register in the write-back stage. *Figure 7* shows a block diagram of the execution unit. Since the PLCRISC supports both 16 bit and 32 bit operations for all arithmetic logic instructions, swap multiplexer logic is implemented between the register file and function units.

In the PLCRISC for fast execution of bit instructions, a bit processing unit is implemented. The bit processing unit consists of two accumulators and a bit manipulation logic. The accumulators are implemented as stacks called 'logic unit stack' (LUS) and 'master control stack' (MCS). The LUS is used to keep each bit instruction result, and the MCS is used for master control operations of the LD language programs. Since the MCS accumulator has a 32 bit shift register, the master control operation can be nested 32 times. The bit manipulation logic carries out bit logic operations such as AND, OR, XOR and NOT. The unit also receives the result of a word compare instruction and provides the execution condition code for the conditional executions of word instructions.

Program counter unit

The PCU is based on a program counter register (PCR) that points to the instruction to be executed. The value of the PCR is added by four for a normal operation and can be changed

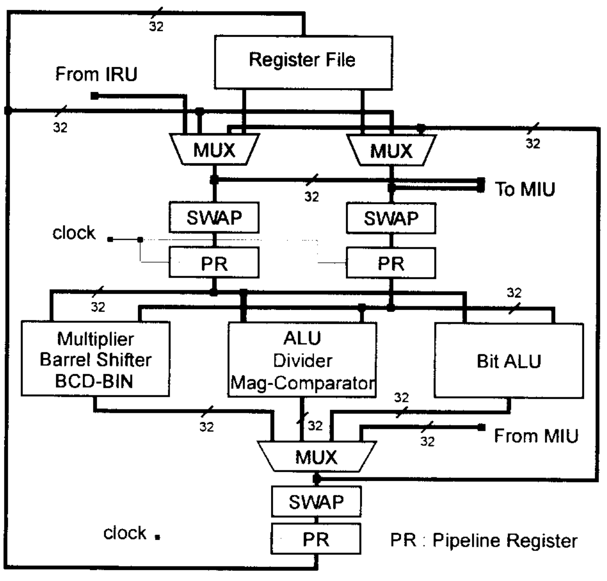


Figure 7 Block diagram of the execution unit

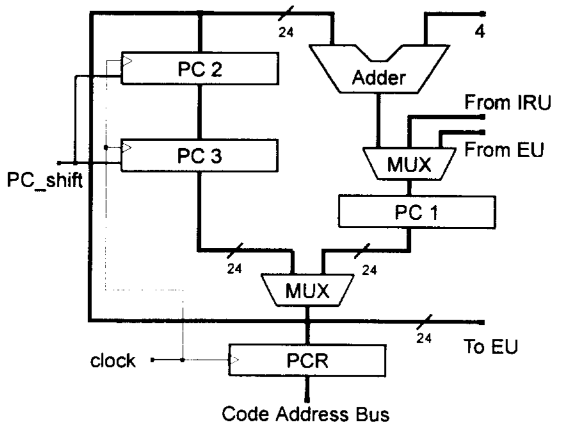


Figure 8 Block diagram of the program counter unit

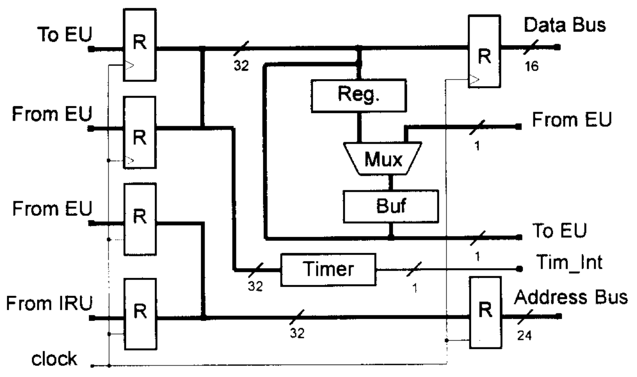


Figure 9 Block diagram of the memory interface unit

by branch instructions. This unit also works with the IRU's exception unit to handle exceptions and interrupts.

During normal operation, the instruction addresses in the instruction decoding stage and the memory fetch and execution stage are saved in the PCU's program counter chain, which consists of two cascaded 32 bit registers. The PCU contains a mapping table for exception service routines. When an exception or an interrupt occurs, the instruction in the instruction fetch stage is flushed and the address of the instruction is saved. Then the PCR loads the address of the interrupt vector table and jumps to the start address of the interrupt service routine. During the execution of the service routine, the PC chain is frozen.

In the PLCRISC, the exception check and external interrupt detection are accomplished in the early stage of an instruction execution. To support this mechanism, several hardware resources are added, such as a zero count check logic, a memory boundary check logic, a division by zero check logic, and a binary-to-BCD overflow check logic. These logics reduce the complexity of exception processing, which is one of the most difficult aspects of designing RISC processors. The early detection of exception condition prevents unnecessary instructions from being fetched.

In returning from the interrupt service routine, a return from the exception instruction enables the program counter chain again and successively shifts out the two addresses contained in it. The program then can execute the instructions that were flushed when the exception or interrupt occurred. After these operations, the program enables the exceptions and interrupts again.

Memory interface unit

In this unit, an interface between the internal data path and the external data memory is implemented. To meet the high memory bandwidth typical of PLCs (see *Table 3*), the PLCRISC has separate buses for instructions and operands, and thus fetching of an operand in each instruction in the MIU is performed concurrently with an instruction fetch in the IRU.

The bit instructions need bitwise data and must access the memory bitwisely. But usually in PLCs, the main memory is made to have a byte or word boundary structure due to the design complexity and cost. Therefore, to overcome the incompatibility of the two addressing modes, the MIU contains a memory interface logic and a multiplexer. For a bit output instruction, the memory interface logic has a buffer that holds the 16 bit data which was read in the previous instruction. When an output instruction is executed, the multiplexer substitutes the stored data with resulting bit, and the 16 bit value is written into the memory.

The MIU also implements an internal timer. Because PLC applications usually entail periodic operation and, thus, a timer instruction, the PLCRISC provides one internal timer avoiding the need for an extra timer chip. The internal timer in the MIU generates an exception to the IRU for a timer interrupt operation. Several instructions are defined to control the timer and load a counter value.

PERFORMANCE EVALUATION AND VLSI DESIGN

Usually the performance of a PLC is calculated and compared to that of other models with just the execution time of bit instructions. This concept is similar to the MIPS of general purpose processors. However, such a method is somewhat far from a correct performance index. In a commercial PLC, word instructions take a longer time than do bit instructions, although bit instructions are more frequently used. To calculate the performance more accurately, the performance must be evaluated by the execution time of mixed PLC instructions. In doing so, because the PLCRISC has simple word instructions, the word instructions of a PLC program must be translated into several PLCRISC instructions. *Table 5* shows translation examples of several representative PLC instructions.

Based on the execution times of mixed PLC bit and word instructions which are translated into PLCRISC instructions, the performance of the PLCRISC is obtained by timing simulations in a CAD environment as shown in *Figure 10*. The more word instructions in a PLC program that are used, the more PLCRISC instructions that are needed. Thus, the performance is degraded. This result shows that the PLCRISC is much faster than existing PLCs; for example, 1 kstep/ms for the Mitsubishi A series and 1.67 kstep/ms for the Siemens S5-135U[14]. In the simulation, the PLCRISC is assumed to be running at a 16 MHz clock.

Table 5 Examples of PLC instruction translation

PLC instructions	Converted PLCRISC instructions	
str r001	str r001	; start rung
out r001	iniout r001	; init out (read data)
	out r001	; out (modified write)
	load.l r0,w2	; load a word from memory
	load.l r1,w3	; load a word from memory
add w1,w2,w3	add.w r1,r2	; add word data
(w1=w2+w3)	store.l w0,psw	; store PSW to memory
	store.l w1,r2	; store the result to memory

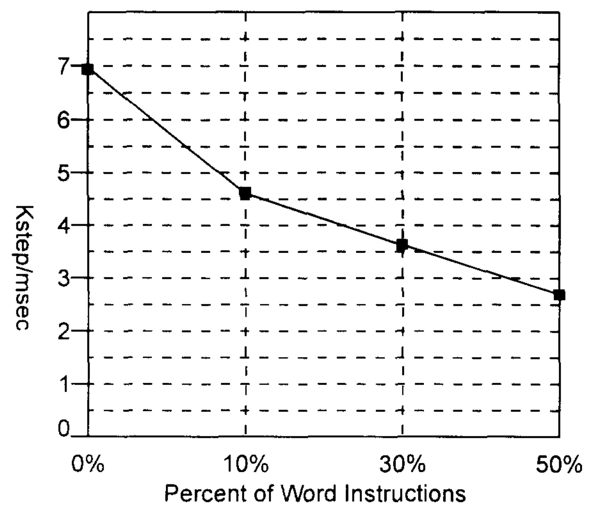


Figure 10 Performance of the PLCRISC

The unit of the performance is ksteps/ms, in which 'step' means the number of executed instructions. With the performance index, the user can determine how many I/O points the PLC processor can handle within a finite scan time.

The PLCRISC is developed in ASIC form and implemented by VHDL (VHSIC Hardware Description Language). Because VHDL is a design tool that uses a high level language, the design time can be reduced and the design is maintained efficiently. Recently, VHDL, which was designed mainly as a simulation language, has been used effectively as a synthesis language[15].

In the design of the PLCRISC, after the functionalities of the processor are simulated in the VHDL environment, the circuits are made by VHDL synthesis. In the early stages of the implementation, several blocks are divided according to their functions. Among them the internal control part is made by VHDL synthesis and the data path parts, including the ALU, the multiplier and the barrel shifter, are implemented by a data path compiler of the ASIC vendor.

To prove the validation of the operations of the PLCRISC, a software simulator, named by PLCSIM, is implemented in parallel with the processor design. The PLCSIM has a graphical user interface based on an X-window system. After the PLCSIM reads and executes the PLCRISC programs, it produces the expected state of the processor at the end of every instruction cycle. The results of the simulator are compared with the simulation results of the CAD tool to verify the functionality of the PLCRISC. The simulator has an ability to randomize the data value referred by the instructions and also the ability to debug the entire internal states of the processor. Thus, it can be used as a software development tool for PLC programming.

CONCLUSION

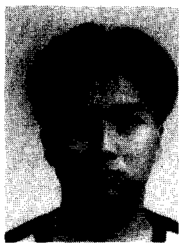
In this paper, a special purpose RISC microprocessor for programmable logic controllers is proposed. Through the investigation of the features required for high performance PLC processors and the analysis of existing PLC programs, an instruction set is selected and the basic architecture is suggested. Specifically, PLCRISC instructions significantly reduce the overhead that arises when general purpose microprocessors, which are designed for word instructions, are used for PLCs, which predominantly operate with bit instructions. The PLCRISC employs a four-stage pipeline, separated external buses, and special hardware support for the interaction between bit and word instructions, in order to enhance the performance. Thus, it can execute PLC instructions very fast and efficiently. The PLCRISC satisfies the increasing needs of functionality and speed in industrial sequential control fields and can be adopted in a large, high-performance and high-speed PLCs.

The proposed PLCRISC processor has been evaluated through simulation using VHDL. The simulation result shows that this processor runs much faster than the microprocessors used in commercial PLCs. The PLCRISC is under fabrication in a 0.8 μm. CMOS technology, and is shortly scheduled to be implemented in a real commercial product.

REFERENCES

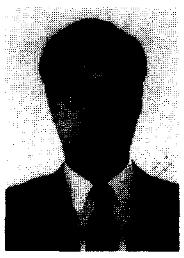
1 Warnock, I G *Programmable Controllers: Operation and Application*, Prentice-Hall, Englewood Cliffs, NJ (1988)
2 Kim, Jong-il, Park, J and Kwon, W H 'Architecture of a ladder solving processor for programmable controllers' *Microprocessors Microsyst.* Vol 16 No 7 (1992) pp 369–379
3 Murakoshi, H, Sugiyama, M, Ding, G, Oumi, T, Sekiguchi, T and Dohi, Y 'A high speed programmable controller based on Petri net' *Proc. IECON* (1991) pp 1966–1971
4 Hennessy, J L 'VLSI processor architecture' *IEEE Trans. Comput.* Vol C33 No 12 (1985) pp 1221–1246
5 Stallings, W 'Reduced instruction set computer architecture', *Proc. IEEE* Vol 76 No 1 (1988) pp 58–75
6 Shimokawa, Y, Matsushita, T, Furuno, H and Shimanuki, Y 'A high-performance VLSI chip of programmable controller and its language for instrumentation and electric control' *Proc. IECON* (1991) pp 884–889.
7 *Programmable Controllers – Part 3: Programming Languages*, IEC 1131-3, IEC (1993)
8 *Samsung BRAIN SPC-300 User's Manual*, Samsung Aerospace Industries (1990)
9 Koo, Kyeong-hoon, Rho, G S and Kwon, W H 'An architecture of the RISC processor for programmable controllers' Technical Report 94-01, Information Systems Laboratory, SNU, submitted to International Conference on Industrial Electronics
10 Hennessy J L and Patterson D A *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann (1990)
11 Howard, E and Irissou, B 'A Mips R2000 Implementation' *IEEE Comput. Fut.* (1991) pp 46–54
12 Gray, J, Lenell, J, Naylorr, A and Bagherzadeh, N 'Design and implementation of the 'Tiny RISC' microprocessor' *Microprocessors Microsyst.* Vol 16 No 4 (1992) pp 187–193
13 Anido, M L and Allerton, D J 'RISC design for computer image generation' *Microprocessors Microsyst.* Vol 14 No 6 (1990) pp 341–350
14 'PLCs & positioning systems' *Drives Contr.* Vol 9 No 6 (July/August 1993)
15 Camposano, R, Saunders L F and Tabet, R M 'VHDL as input for high-level synthesis' *IEEE Des. Test Comput.* (1991) pp 43–49

Gab Seon Rho was born in Seoul, Korea on 10 February 1965. He received BS and MS degrees in control and instrumentation engineering from Seoul National University in 1988 and 1990, respectively. He is currently a PhD student in the Department of Control and Instrumentation Engineering, Seoul National University. His main research interests are computer applications for factory automation, computer architecture, real-time systems and discrete event systems.

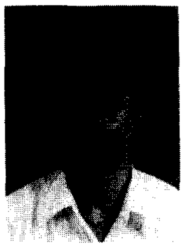


Kyeong-hoon Koo was born in Seoul, Korea on 12 September 1967. He received BS and MS degrees in control and instrumentation engineering from Seoul National University in 1990 and 1992 respectively. He is currently a PhD student in the Department of Control and Instrumentation Engineering, Seoul National University. His main research interests are computer applications for factory automation and real-time control systems.

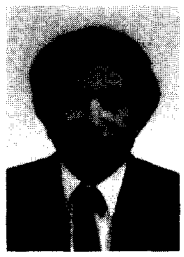
Naehyuck Chang was born in Seoul, Korea on 19 March 1967. He received BS and MS degrees in control and instrumentation engineering from Seoul National University in 1989 and 1992, respectively. He is currently a PhD student in the Department of Control and Instrumentation Engineering, Seoul National University. His main research interests are computer applications for factory automation, computer graphics, real-time systems and discrete event systems.



Jaehyun Park was born in Seoul, Korea on 8 October 1963. He received BS, MS and PhD degrees, all in control and instrumentation engineering, from Seoul National University in 1986, 1988 and 1993, respectively. He is currently a research fellow in the Electrical Engineering and Computer Science Department of the University of Michigan. He and his colleagues are currently building a mesh multicomputer, called HARTS, and an open architecture machining controller, called UMOAC. His main research interests are computer architectures for real-time control systems, computer applications for factory automation, and discrete event systems.



Yeong-gi Kim was born in Korea, 1963. He received a BS degree in electronic engineering from Kyeong-buk National University, Daegoo, Korea in 1986. Since 1986, he has been at Samsung Aerospace Industries, Ltd., Korea and is currently a senior engineer. He and his colleagues are now developing a medium-scale PLC. His current interests include industrial automation application and digital system design.



Wook Hyun Kwon was born in Korea on 19 January 1943. He received BS and MS degrees in electrical engineering from Seoul National University, Seoul, Korea, in 1966 and 1972, respectively. He received a PhD in control from Brown University in 1975. From 1975 to 1976, he was a research associate at Brown University. From 1976 to 1977, he was an adjunct professor at University of Iowa. Since 1977, he has been with Seoul National University, now as a professor. From 1981 to 1982, he was a visiting assistant professor at Stanford University. Since 1991, he has been the director of Engineering Research Center of Advanced Control and Instrumentation. His main research interests are currently multivariable robust and predictive controls, discrete event system, automation network and computer applications for factory automation.