고속 프로그램형 논리 제어기 구현을 위한 래더 다이어그램 해석 방법

A Translation Method of Ladder Diagram for High-Speed Programmable Logic Controller

김 형 석, 장 래 혁, 권 욱 현
(Hyung Suk Kim, Nae Hyuck Chang, and Wook Hyun Kwon)

Abstract : This paper proposes a translation approach for PLCs (Programmable logic controllers) converting ladder diagrams directly to native codes, and describes detailed steps of the method followed by performance evaluation.A general-purpose DSP (Digital signal processor) based implementation validates the approach as well. A benchmark test shows that the proposed translation framework fairly speeds up execution in comparison with the existing interpretation approach.

Keywords : PLC(programmable logic controller), LD(ladder diagram), translation method, DSP

I. 서론

프로그램형 논리 제어기(programmable logic controller, 이하 PLC)는 계전기 회로(electromagnetic relay circuit)를 사용하는 순차 제어기를 교체하기 위해 개발되었는데, 논리, 순차, 타이밍, 카운트, 연산 등을 기본으로 하는 제어 시스템 구현에 현재 널리 사용되고 있으며, 그 응용 분야는 조립 공정, 화학 공정, 운송 시스템, 에너지 시스템 등 자동화를 위한 디지털 시스템 제어기의 여러 분야에 이른다[1]. 자동화 시스템이 더욱 발전함에 따라, PLC가 처리해야 하는 작업들은 더욱 더 복잡해져 왔으며, 이러한 작업들을 보다 능률적으로 처리하기 위해서, PLC는 기본 래더 다이어그램(Ladder Diagram, 이하 LD) 외에 100여종 이상의 다양한 응용 명령어를 보유하게 되었고, SFC(Sequential Function Chart), IL(Instruction List), FBD(Function Block Diagram), ST(Structured Text) 등과 같은 PLC 표준 언어를 지원하게 되었다. 이러한 언어들은 래더 언어의 수행 방법을 근간으로 하며, 프로그램 작성시의 복잡도를 덜어준다[2].

PLC의 성능 척도에 있어 수행 속도가 무엇보다도 중요한 요인으로, 수행하여야 하는 작업이 복잡해짐에 따라 수행 속도에 대한 요구 사항은 고성능의 마이크로 프로세서를 손쉽게 사용할 수 있는 현재에도 여전히 해결해야 할 과제로 남아 있다. PLC가 발전함에 따라서 그 적용 범위 또한 넓어져서, PLC 프로그램에서 복잡한 실수 계산 등 연산 시간이 오래 걸리는 응용 명령어가 차지하는 비중도 높아지고 있다. PLC 언어의 특성상 흐름 제어, 단순 메모리 입출력 등이 복잡한 응용 명령어와 긴밀히 결합되어, 일반 컴퓨터 응용보다 프로그램 수행 속도의 증가는 용이하지 않다[1].

PLC의 수행 속도를 증가시키는 방법에는 하드웨어를 개선하는 방법, 소프트웨어를 개선하는 방법 등 여러 가지가 있을 수 있다. 하드웨어를 개선하는 접근 방법 중의 하나로, 새로운 구조의 연산 프로세서에 관한 연구가 계속되어 왔다. 즉, 병렬 처리 기법을 이용하거나[3][4] 전용 집적회로(ASIC) 설계 기법을 이용하여[6]-[8] PLC 전용의 프로세서를 개발하여 성능 향상을 추구한다. 이와 같은 기법은 하드웨어 자원을 투자하여 프로세서 성능을 근본적으로 개선하는 접근 방법이 되나 비용 면에서 불리하며, 전용 하드웨어에의 의존도가 높아 확장 및 변경이 용이하지 않다는 단점을 수반한다.

래더 명령어 중에서 논리 처리 명령의 비중이 크다는 특징을 활용하여 배열 프로세서(array processor) 구조를 채택한 연구[5]는 동시에 여러 논리 명령어를 처리함으로써 논리 명령어의 수행 속도를 획기적으로 증가시켰으나, 논리 명령과 응용 명령 간의 유기적 관계나 응용 명령 수행 자체에는 해결책을 주지 않아서, 높은 비용, 응용 명령어 구현의 방법의 어려움 등이 단점으로 작용하여 실제적인 측면에서 불리하다. 보다 현실적인 방법으로, 명령 축약형 프로세서(RISC)에 관한 연구들[6][7]은 범용 프로세서에 PLC의 논리 명령어와 기본적인 응용 명령어를 수행하는 명령어를 추가하였다. 이러한 프로세서는 PLC의 수행 속도를 증가시키는 가장 효과적인 방법이 된다. 그러나, 빠르게 발전하는 복잡한 응용 명령어들을 추가로 수용해야 할 문제와 가격 대 성능비가 눈부시게 발전하는 범용의 마이크로 프로세서와는 달리 짧은 기간 내에 추가 설계, 제작하는데 매우 불리하므로, 빠르게 변화하는 수요를 충족시키는데 어려움이 발생한다.

범용 마이크로 프로세서를 사용하는 PLC의 경우에는 전용 프로세서를 사용하는 PLC에 비하여 실행 속도 면에서는 다소 불리하나[6], 일반적으로 성능의 향상 주기가 특수 프로세서에 비해 빠르고 값싼 비용으로 빠른 기간 내에 설계, 제작할 수 있는 장점이 있다. 그리고, 성능의 향상 방법이 운영 체제와 그 내부의 명령어 해석, 실행 방법의 개선 등 소프트웨어에 의하므로 비용 면에서 매우 유리할 뿐 아니라 확장성과 변경의 용이성도 동

접수일자 : 1998. 6. 11., 수정완료 : 1998. 11. 26.
김형석, 권욱현 : 서울대학교 전기공학부
장래혁 : 서울대학교 컴퓨터공학과

시에 제공한다. 그러나, 범용의 마이크로 프로세서를 사용하는 경우에는 PLC 명령어와 마이크로프로세서의 명령어에 내재하는 문법의 격차에서 많은 성능의 손실이 발생한다. 현재까지 널리 사용되는 해석 방법은 이와 같은 문법 격차에서 오는 성능 손실을 크게 고려하지 않는다. 그러나, 과거의 프로세서와는 달리 파이프라인 구조를 사용하는 고성능 프로세서의 경우, 성능 손실은 더욱 더 크게 증폭된다.

또한, PLC 등의 제어 시스템에는 진동, 전자파 등의 열악한 환경 하에서 그 신뢰도가 낮은 자기 디스크 등에 해석기 프로그램이 내재해 있을 수 없으며, 대신에 용량이 적은 메모리를 사용하여야 한다. 따라서, 메모리에 PLC 프로그램 해석기를 내재하자면 알고리듬이 복잡하지 않아 프로그램의 길이가 작은 해석 방식이 유리하기 때문에 미리 코딩된 블록으로 온라인 상에서 교체되어 실행되는 방식이 현재까지 이용되고 있다. 그러나, 복잡한 응용 명령의 추가와 대용량의 PLC의 등장으로 인해 더 이상 짧은 길이의 해석기 프로그램으로는 적합한 동작을 구현할 수 없게 되었다.

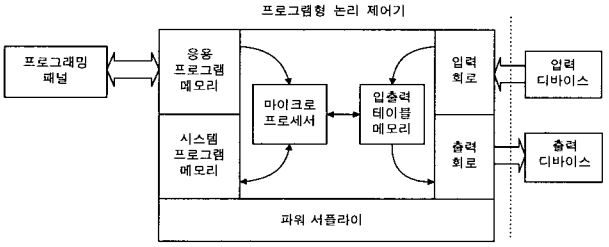


그림 1. PLC의 일반적인 구조.
Fig. 1. Common structure of PLC.

전술한 문제들에 대해 본 논문은 이를 극복하는 해석 방법을 제안하고, 그 구조를 구체적으로 서술할 것이다. 내용과 구성은 다음과 같이 서술된다. 2장에서는 PLC에서의 일반적인 LD의 해석 방법을 서술하고 3장에서는 성능을 향상시킬 수 있는 LD의 해석과 실행을 위한 구조를 제안한다. 또한, 실험을 위하여 제작한 PLC 시스템에 대해 간단히 언급하며 4장에서 실험을 통한 평가로 두 방식 간의 성능 차이를 비교, 분석한다. 5장에서는 결론과 함께 앞으로의 추세 등을 언급하고 끝을 맺는다.

Ⅱ. LD 해석기의 일반 구조

PLC의 일반적인 구조는 그림 1에서 보여진다. 제어 유닛은 입력 장치를 통해서 제어 대상의 상태를 입출력 테이블 메모리(또는 이미지 메모리)로 옮겨오고, 시스템 프로그램이 응용 프로그램 메모리 내의 응용 프로그램을 해석, 실행하여 생성된 값을 출력 장치를 통하여 제어 대상에 전달한다.

LD는 주요 PLC들에서 공통적으로 사용되는 기호와 용어들을 고려하여 IEC(International Electrotechnical Commission)에 의해 개발되었다[9]. 이 프로그램 방법은 릴레이를 사용하여 논리식을 설계하기 위해 사용된다. 그림 2는 IEC 1131-3의 표준에 기반한 LD 프로그램의

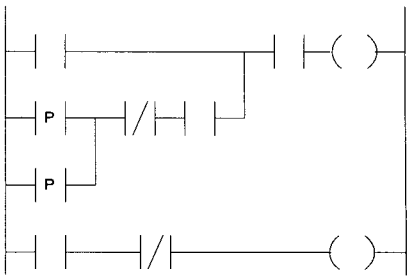
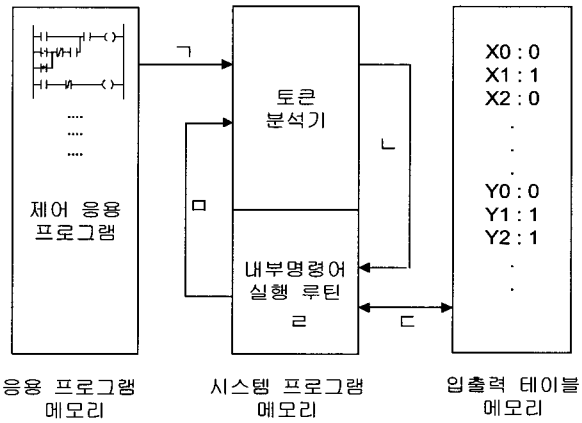


그림 2. LD의 예.
Fig. 2. an example of LD.

예이다. 이러한 LD 프로그램을 해석하는 방법은 여러 가지가 있겠으나 그 대체적인 구조는 그림 3과 같다. 이 그림은 응용 프로그램 메모리와 시스템 프로그램 메모리에 저장되어 있는 프로그램, 입출력 테이블 메모리 간의 데이터 교환과 일련의 동작 원리를 보여주며, PLC가 제어 동작을 시작하면 그림의 (ㄱ)-(ㅁ)의 동작을 반복한다.



ㄱ) 응용 프로그램 메모리에서 명령어 단위로 읽어오고 해독한다.
ㄴ) 각 명령어와 대응하는 실행 루틴을 호출한다.
ㄷ) 오퍼랜드(operand)를 해석하고 입출력 테이블 메모리에서 데이타를 읽는다.
ㄹ) 지정된 동작을 수행한다.
ㅁ) ㄱ)으로 돌아가서 반복한다.

그림 3. LD 프로그램의 해석 과정.
Fig. 3. Steps of LD program translation.

PLC는 제어기기 형태로, 내장된 사용자 인터페이스를 가지고 있지 않은 것이 일반적이다, 대신에 계산기나 컴퓨터의 자판과 흡사한 프로그래밍 패널이 통신 포트를 통하여 연결되어 있다. PLC에서의 프로그래머는 직접 래더 다이어그램을 그리거나 텍스트 형태의 명령어 리스트를 입력하여 응용 프로그램을 구성하고 응용 프로그램 메모리에 이를 저장한다. 그러면 PLC는 이 프로그램을 중간 코드로 변환하고 자신의 메모리에 로드시키게 된다. 그 후에 PLC 제어기를 동작시키게 되면, 프로그램형 논리 제어기 내부의 명령어 번역기가 중간 코드를 하나씩 가져와서 이를 해독한 후 그 명령을 수행하는 서브루틴을 호출하여 그 동작을 수행하게 된다. 그림 3에서 보여졌듯이 기존의 해석 방식에서는 각 내부 명령어(mnemonic)의 실행을 위한 서브루틴들이 있어서 이를 호출하

여 이용하기 때문에 여러 번의 호출(call), 회귀(return)가 필요하게 된다. 보통 고성능 RISC 프로세서에서도 이런 동작들은 긴 시간이 소비된다.

기존의 해석 방식은 제어 동작 중에 해석, 실행이 되기 때문에 제어기 내부에 저장된 응용 프로그램의 원시 코드를 하나씩 읽고 해석하여 미리 작성된 매크로 루틴으로 대체하여 수행할 수 밖에 없어 수행 속도가 저하될 뿐더러, 최근에 추가되는 실행 시간이 매우 긴 특수 연산의 경우 시간의 지연이 크게 되어 이 두 가지 시간 지연 요소의 조합으로 인해 PLC의 수행 성능의 향상을 기대할 수 없게 된다. 따라서, 이러한 해석 방식의 느린 속도와 비효율성을 극복하기 위한 효율적이고 고속인 해석과 실행 방식이 필요하게 된다.

Ⅲ. LD 해석기의 구현

1. 기본 구조

그림 4는 LD 응용 프로그램이 해석되어 실행 코드로 변환되는 과정을 나타낸 것이다.

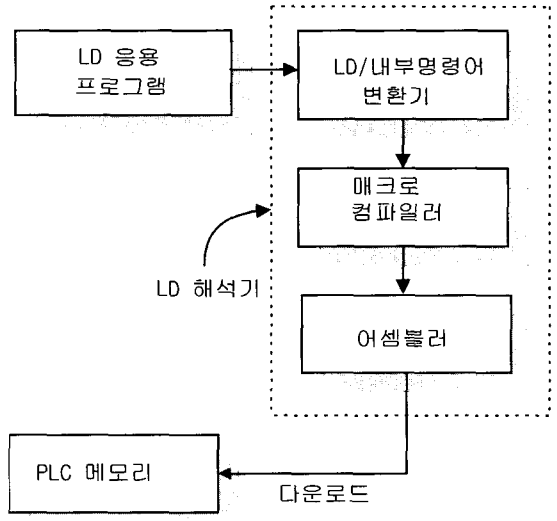


그림 4. LD 해석기의 구조.
Fig. 4. The structure of LD translator.

첫번째로, LD 응용 프로그램이 LD/내부 명령어 변환기에서 내부 명령어로 이루어진 프로그램으로 변환된다. PLC는 LD를 첫 번째 단(rung)부터 마지막 단까지 연속적으로 모든 입력값들을 조사한 후에야 출력값이 계산된다. 그림 5의 (ㄴ)은 LD를 해석하기 용이하도록 변환하기 위한 나무(tree) 구조로 (1)과 같은 논리식을 블록 단위로 나타낸 것이다.

$$\text{출력 (ON/OFF)} = 1 \wedge [2 \vee [3 \wedge 4]] \qquad (1)$$

$\wedge$: 논리곱, $\vee$: 논리합

그림 5의 (ㄷ)은 변환된 결과인 내부 명령어이다. LOAD라는 내부 명령어는 오퍼랜드로 넘겨받은 접점의 값(On/Off)을 스택에 차례대로 저장해 놓는 동작을 한다. AND (또는 OR)는 스택에서 직전에 저장된 접점값을 꺼내어 오퍼랜드와 $\wedge$(또는 $\vee$) 로직을 수행한다. ANDBLK, ORBLK은 스택에 저장된 최근 두 값에 대해서 각각 $\wedge$, $\vee$ 로직을 수행하며, 이에 앞서 AND, OR로 그 논리값이 결산된 블록들간의 계산을 목적으로 한다. 이러한 계산을 위하여 스택이 필요한데, 그림 6의 R10과 같은 범용 레지스터를 사용하였다. 이 레지스터는 다른 목적으로 그 값이 바뀌어서는 안된다.

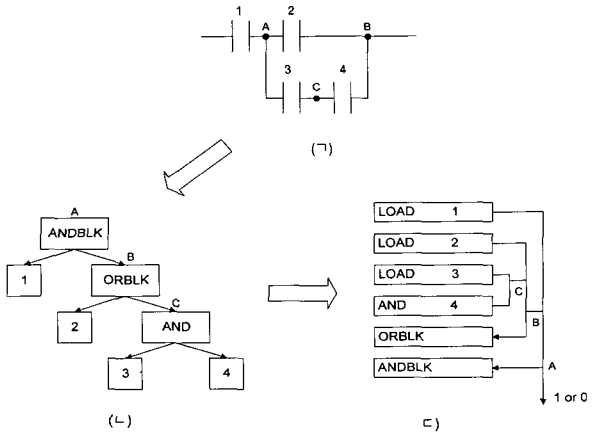


그림 5. LD의 명령어 내부 명령어 리스트 변환.
Fig. 5. Translation of LD into mnemonic list.

이렇게 텍스트 형태로 바뀐 LD는 일반적인 프로세서의 어셈블리 명령어와 그 형태가 가깝기 때문에 어느 정도 고정된 형식을 갖게 되어 어휘 분석(lexical analysis) 과정이 수월해지게 된다.

매크로 컴파일러(macro compiler)는 내부 명령어를 상응하는 미리 짜여진 어셈블리 코드로 대치 변환한다. 이 작업 중에 프로그래머에게 원시 코드(source code)에 대한 에러들을 알려 주며, 또한 기계어로 변환시 그 길이나 실행 속도 등의 전체적 효율을 증대하기 위해 코드 최적화를 실행한다. 이 과정은 내부 명령어를 입력받아서 출력으로 각 마이크로 프로세서에 특정한 어셈블리 코드를 만들어내는 작업을 수행한다. 결과물인 어셈블리 코드는 다음 단계인 어셈블러에 입력된다. 그림 6에서 전 단계의 내부 명령어 리스트에 대한 출력인 어셈블리 코드를 나열한다. 어셈블리 코드는 TMS320C40 DSP의 코드이다.

```
;M_LDID   0ffffh,0ffffh,R11
      LDHI    0ffffh,R11
      OR      0ffffh,R11
;; LD X1
;M_LDIA   08000h,01h,R0
      LDPK    08000h
      LDI     @01h,R0
      AND     R11,R0
      LSH     -1,R10
      OR      R0,R10
      BR      LD001
LD001:
;; LD X2
;M_LDIA   08000h,02h,R0
      LDPK    08000h
      LDI     @02h,R0
      AND     R11,R0
      LSH     -1,R10
      OR      R0,R10
      BR      LD002
LD002:
;; LD X3
;M_LDIA   08000h,03h,R0
      LDPK    08000h
      LDI     @03h,R0
      AND     R11,R0
      LSH     -1,R10
      OR      R0,R10
      BR      LD003
LD003:
;; AND X4
;M_LDIA   08000h,04h,R0
      LDPK    08000h
      LDI     @04h,R0
      AND     R11,R0
;M_LDID   07ffffh,0ffffh,R1
      LDHI    07ffffh,R1
      OR      0ffffh,R1
      OR      R1,R0
      AND     R0,R10
      BR      AND001
AND001:
;; ORB
      LDI     R10,R0
      LSH     1,R10
;M_LDID   08000h,0h,R1
      LDHI    08000h,R1
      OR      0h,R1
      AND     R1,R0
      OR      R0,R10
;; ANDB
      LDI     R10,R0
      LSH     1,R10
;M_LDID   07ffffh,0ffffh,R1
      LDHI    07ffffh,R1
      OR      0ffffh,R1
      OR      R1,R0
      AND     R0,R10
```

그림 6. 변환된 어셈블리 코드.
Fig. 6. Converted assembly code.

다시 이것을 어셈블러 프로그램이 PLC의 프로세서에서 곧바로 실행할 수 있는 기계어로 변환시키는데, PLC 프로그램은 대부분의 명령어들이 각 명령어 별로 독립적

인 관계를 가지므로 각 명령어 블록 간에 상관 관계 있는 것들(라벨, 변수 등)을 먼저 처리한 후에 명령어 단위로 끊어서 차례로 어셈블러를 통과시키면 메모리 절감과 해석 속도 면에서 향상된 성능을 볼 수 있다.

일반적인 어셈블러는 2 패스(pass) 구조로, 첫 패스에서 먼저 LD001, AND001 등의 라벨에 대해 대응되는 어드레스와 수치 등을 저장하는 심벌 테이블(symbol table)을 구성한다. 두 번째 패스에서 이 심벌 테이블을 참조하며 기계 코드로 변환한다. PLC는 논리 계산으로 실행 여부를 결정하는 동작 특성상 어셈블리 코드 내에 분기가 많이 존재하여 라벨 처리에 많은 메모리를 할당할 수가 있다. 그런데, 래더 다이어그램의 한 노드(node)는 그림 7처럼 하나의 블록 단위로 어셈블리 코드가 나뉘어지고 몇 개의 흐름 제어 명령어를 제외하면 대부분이 다른 노드와 공통적인 심벌을 가지지 않는 특성이 있다. 그림 7은 이 특성을 이용하여 구성한 메모리 사용량이 적고 속도가 빠른 어셈블러의 구조를 나타낸 것이다. 먼저 루프나 분기 등의 PLC 명령어에서 공통적으로 사용되는 몇 개의 라벨들을 1단계에서 테이블에 저장하고, 2단계부터는 (2, 3), (4, 5), (6, 7)...과 같이 블록 단위로 일반적 2패스 어셈블러의 동작을 반복한다. 하나의 블록의 변환이 끝날 때마다 테이블에서 1에서 저장했던 전역 라벨을 제외하고 모두 삭제하면 메모리의 절감과 속도의 향상을 가져오게 된다.

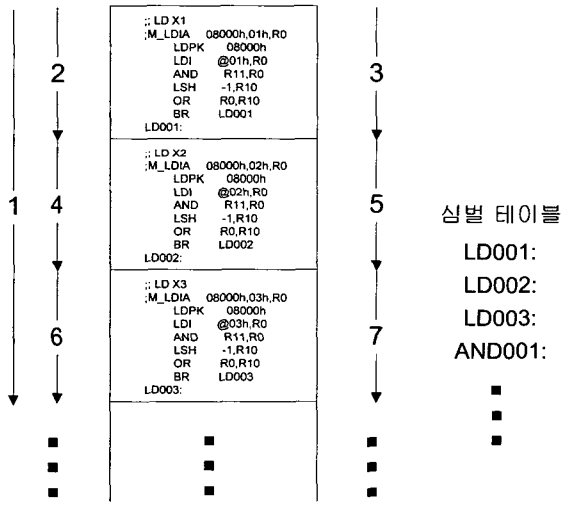


그림 7. 어셈블러의 구조.
Fig. 7. Structure of the assembler.

로더의 최종 출력인 목적 코드(object code)는 PLC가 플랜트의 제어 동작을 하고 있지 않은 오프라인(off-line) 시에 PLC 내의 메모리로 전송된다. 이를 실행시키면 PLC는 이 코드를 순차적으로 실행하여 플랜트 제어를 하게 된다. 여기서 강조될 것은 프로그래머가 로더를 이용하여 작성한 래더 프로그램은 중간 코드(intermediate code)의 형식으로 PLC의 메모리에 전송되어 인터프리트(interpret) 방식으로 수행되는 것이 아니라 로더에서 컴파일(compile) 방식으로 실행 가능한 이진 코드(binary code)로 바뀐 후에 적절한 방법을 사용하여 이를 전송, 실행한다는 것이다.

이러한 LD 해석의 과정을 단계별로 표 1에서 간략하게 각 입출력 관계와 기능을 토대로 정리하였다.

표 1. LD 해석의 단계별 특성.
Table 1. Feature of steps of LD translation.

	LD/내부명령 어변환	매크로 컴파일러	어셈블러	로더
입력	LD	내부명령어	어셈블리	이진 코드
출력	내부명령어	어셈블리	이진 코드	HEX코드
기능	텍스트 형태로 변환	특정 CPU의 언어로 기능 구현, 최적화	PLC 언어 해석에 적합한 어셈블러	다운로딩, 다운로드 시간 줄임

2. PLC 시스템 제작과 설정

200여개의 각종 기본, 응용 명령어를 지원하며 10,000점 이상을 지원하는 대용량 PLC를 제작하였으며, 프로세서로는 디지털 시그널 프로세서(DSP:Digital Signal Processor) TMS320C40-40을 사용하였으며, 로더와의 통신을 위하여 비동기 직렬 통신 장치를 설치, 연결하였다. DSP의 내부 캐쉬는 비교를 위한 실행 시간을 일정하게 산출하기 위하여 사용하지 않도록 지정하였으며 접점값의 저장을 위한 입출력 테이블 메모리는 중속 SRAM을 사용하고 대기 상태(wait state)로 50ns를, 다운로드된 실행 코드 또는 중간 코드와 운영 체제의 저장 영역으로는 빠른 실행을 위해 고속 SRAM을 사용하고 대기 상태는 0으로 설정하였다.

Ⅳ. 성능 평가

구현된 해석기의 제어 응용 프로그램 해석 속도를 비교하여 보면 기존의 해석 방식이 제안된 해석 방식에 비해 그 해석 속도가 빠르다. 왜냐하면 제안된 해석 방식은 컴파일 과정과 어셈블 과정을 거칠 뿐만 아니라 파일 입출력을 여러 번 하기 때문에 미리 명령어에 대한 변환 루틴이 준비되어 있는 기존의 해석 방식에 비해 그 속도는 현저히 느리다. 그러나, 이 방식의 프로그램은 제어를 위한 응용 프로그램을 실행하고 있을 때 해석하는 것이 아니라 실행 전에 이미 그 해석을 모두 마치고 실행 가능한 결과 코드를 제어기에 넘겨주기 때문으로, 그 해석에 걸리는 시간이 PLC의 동작시의 성능에는 관련이 없다. 따라서, 두 해석 방식의 비교를 위해서는 해석에 지연되는 시간이 아닌 제어 동작 시, 즉 실행 시간의 비교를 행하는 것이 효과적인 비교이다.

실행 시간 비교를 위하여 제작된 PLC에서 각 명령어의 수행 시간을 측정하기 위한 타이머로는 프로세서에 내장된 것을 사용하였고, 이러한 동일 환경 하에서 각 방식에 대한 비교 분석을 행하였다. 각 명령어 종류별 해석 방식의 교체로 인한 평균 실행 시간의 변화는 그림 8에서 보여진다. 명령어의 종류 구별은 그 내용과 구조가 비슷하여 실행 시간과 그 코드량도 크게 다르지 않으리라 예측되는 것을 같은 종류로 모은 것이다[10,11]. 종

류별로 각 명령어에 대해서 계산된 실행 시간 값을 참고로 하여 기존 해석 방식을 적용한 실행 시간과 제안된 방식을 적용한 실행 시간이 각각 계산되어 있다.

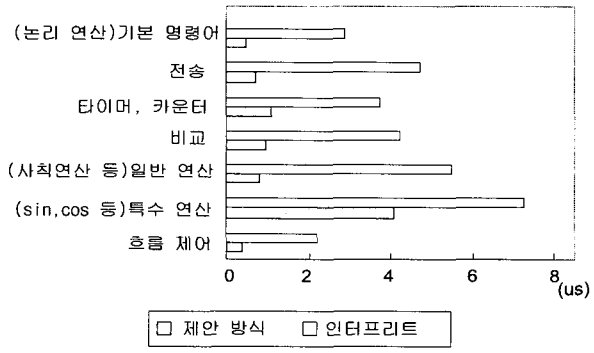


그림 8. 각 명령어별 실행 시간 변화.
Fig. 8. Change of execution time of each mnemonics.

일반 연산 명령어가 실행 시간 변화율이 가장 큰 것을 볼 수 있는데, 일반 연산 명령어는 보통 피연산자 2개와 결과를 저장하는 1개로 구성되는 3개의 오퍼랜드(operand)를 가지고 있기 때문이다. 기존의 해석 방식에서의 오퍼랜드 해석부가 최소한 3번의 복잡한 오퍼랜드 해석 루틴을 거치고 또한 중간 코드를 메모리에서 읽는 횟수가 타 명령어들에 비해 많기 때문에, 그 차이가 다른 것에 비해 크게 나타난다. 반면에, 특수 연산 명령어, 즉 삼각 함수나 지수 함수처럼 그 실행 코드가 길고 그 오퍼랜드의 개수는 많지 않을 경우에는 실행 시간 변화율이 다른 것에 비하여 작아진다. 즉, 두 가지의 수치를 종합적으로 고려하여 볼 때, 제안된 해석 방식은 오퍼랜드를 많이 필요로 하고, 그 실행 코드가 크지 않은 경우 그 실행 속도의 향상 효과가 커지게 된다는 것을 알 수 있다.

그런데, 실제 PLC의 동작 시에 실행 시간의 감소 효과를 분석하기 위해서는 동작 중인 응용 프로그램에 들어 있는 내부 명령어들의 분포를 이용하여 계산되어야 한다. 또한 전술한 바와 같이 동 종류별 명령어들은 그 실행 시간이 비슷하여 평균 실행 시간으로 대표될 수 있기 때문에 각 종류별 명령어의 사용 빈도를 이용하여 전체 실행 시간을 계산하는 것이 큰 오차를 주지는 않는다. 여러 개의 실제 공정에서 사용 중인 프로그램들을 분석한 내부 명령어의 분포도는 그림 9와 같으며, 이를 바탕으로 전체 프로그램 내의 명령어 개수, 기본 명령어의 분포 등을 변화시키면서 응용 프로그램의 실행 시간을 분석하였다.

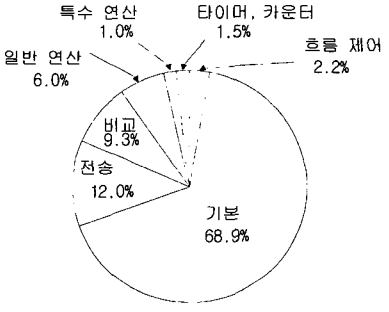


그림 9. 응용 프로그램들에서의 내부 명령어 분포.
Fig. 9. Frequency of mnemonics in application programs.

다음의 (2)는 실험 결과인 기본 명령어의 평균 실행 시간인 $T_{기본명령}$, 기본 명령 이외의 명령어의 종류별 평균 실행 시간인 $T_{응용명령}$과 그림 9의 명령어 빈도 $P_{기본명령}$, $P_{응용명령}$을 이용하여 프로그램 내의 명령어의 전체 개수 N_{exe}를 가정하였을 때의 프로그램 실행 시간을 계산한 것이다. 기본 명령어의 비율 $P_{기본명령}$은 프로그램에 따라 변할 수 있다고 가정하였다.

$$\text{프로그램 평균 실행 시간} = T_{기본명령} \times P_{기본명령} \times N_{exe} + \sum_{응용명령} \left\{ T_{응용명령} \times \frac{P_{응용명령}}{(1-0.689)} \times (1-P_{기본명령}) \times N_{exe} \right\} \quad (2)$$

그림 10은 실행되는 응용 프로그램 내의 명령어의 수(N_{exe})를 300씩 증가시키면서, 즉 응용 프로그램의 길이를 증가시키며 실행 코드 내의 기본 명령어의 비율과 프로그램 실행 시간과의 관계를 나타낸 것이다. 전체적으로 보아 제안된 해석 방식의 적용 시 실행 속도의 상승 효과가 두드러짐을 알 수 있다. 또한, 응용 프로그램 내에서 기본 명령어를 더 많이 사용할수록 그 실행 시간은 감소하는 것을 그림을 통하여 알 수 있다.

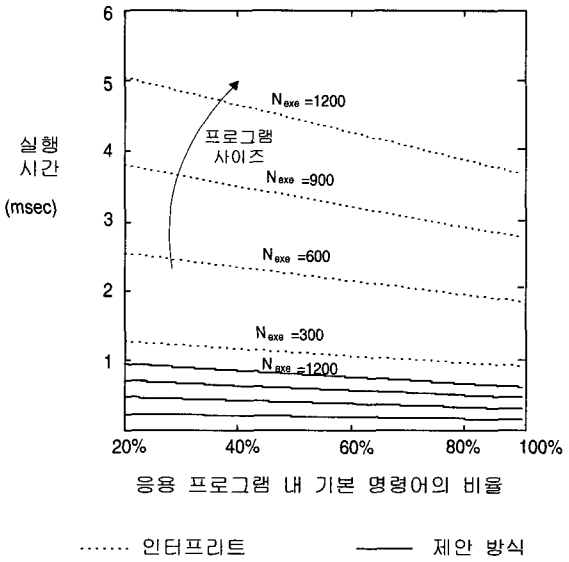


그림 10. 응용 프로그램의 실행 시간 변화.
Fig. 10. Change of execution time of an application program.

제안된 방식은 실행할 이진 코드를 내재할 메모리를 부가적으로 필요로 하기 때문에 메모리 사용량에 대한 연구가 필요하다. 이 측정값을 간략하게 나타내면 (3), (4)와 같다. N_{exe}는 해석될 PLC 응용 프로그램 내의 내부 명령어의 개수를 나타내고, N_{PLC}는 PLC가 지원하는 내부 명령어의 개수를 의미한다.

기존 방식의 평균 메모리 사용량 M'은

$$M' = 2.114N_{exe} + 18.973N_{PLC} + 183 \quad (3)$$

이고, 제안된 방식의 평균 메모리 사용량 M은

$$M = 9.790N_{exe} \quad (4)$$

으로 측정, 계산되었다. 응용 프로그램의 길이가 매우 길어지는 경우 제안된 방식의 메모리 사용량에 부담이 되어질수 있으나, 보통 제어 프로그램은 단순한 프로그램의 반복이기 때문에 많은 영향을 끼치지 않는다.

V. 결론

본 논문에서는 고속을 요구하는 PLC에서 발생하는 프로그램 수행 속도 향상에 대한 문제를 고찰하여 제어 프로그램의 수행 속도를 빠르게 하기 위한 새로운 해석 구조와 실행 방식 등을 제안하고 이를 구현, 실험하여 기존의 방법과의 비교를 통하여 성능을 평가하였다.

제안된 해석 방법은 래더 응용 프로그램으로부터 중간 코드를 만들지 않고 순차적으로 그대로 실행이 가능하도록 한다. 즉 제어기의 운영 체제 내부에 해석부가 내재되지 않은 상태에서 소형 컴퓨터 등의 오프라인 상에서 결과 코드를 생성해내어 제어기에 통신을 통하여 로딩하여 실행시키는 방식이다. 각 방식의 실행 시간을 측정, 비교하기 위하여 PLC를 제작하였는데, 제안된 해석 방식이 기존의 방식에 비해 그 평균 프로그램 실행 시간이 적은 것으로 실험으로 관측되었다. 이것은 호출, 회귀가 거의 없고 기존 방식과 같이 중간 코드 억세스를 위하여 메모리를 참조하지 않기 때문에 산출된 결과로 분석할 수 있다.

PLC의 역할이 확대되어 고속을 요구하고 복잡한 연산의 필요성이 대두되며 내부 명령어의 수가 증가하여 프로그램의 실행 시간과 메모리 사용량이 늘어날 수 있기 때문에 효율적인 해석 방식의 연구는 앞으로 더더욱 중요한 과제가 될 것이다.

김 형 석

1996년 서울대 전기공학부 졸업. 동대학원 석사(1998), 1998년-현재 서울대학교 동대학원 박사과정 재학중. 관심분야는 디지털 시스템 설계, 실시간 시스템, PLC, 필드버스 등.

권 욱 현

1966년 서울대 전기공학과 졸업. 동대학원 석사(1972), 미국 Brown University 제어공학박사(1975). 1976년-1977년 University of Iowa 겸직조교수, 1981년-1982년 Stanford University 객원교수, 1977년-현재 서울대학교 전기공학부 교수, 1991- 현재 제어계측신기술연구센터 소장, 관심분야는 제어 및 시스템 이론, 이산현상 시스템, 산업용 통신망,분산 공정 제어 등.

참고문헌

[1] I. Warnock, *Programmable Controllers - Operation and application*, Prentice Hall, 1988.

[2] International Electrotechnical Commission, *Program-able Controllers - Part 3 : Programming language*, IEC Publication1131-3, 1993.

[3] J. Park, N. Chang, G. S. Rho and W. H. Kwon, "Implementation of a parallel algorithm for event driven programmable dontrollers," *Control Eng. Practice*, vol. 1, no. 4, pp. 663-670, 1993.

[4] J. I. Kim, J. Park and W. H. Kwon, "Architecture of a ladder solving processor for programmable controllers," *Microprocessors and Microsystems*, vol. 16, no. 7, pp. 369-379, 1992.

[5] G. Rho, J. Park, and W. H. Kwon, "Load balancing of a data-flow based programmable controllers," *Proc. of IECON 93, Hawaii, U.S.A*, 1993.

[6] K. Koo, G. Rho, J. Park and W.H. Kwon, and N. Chang, "Architectural design of a RISC processor for programmable logic controller," *Journal of Systems Architecture*, Elsevier, pp. 311-325, no. 44, 1998.

[7] G. Rho, K. Koo, N. Chang, J. Park and W. H. Kwon, "Implementation of a RISC processor for programmable controllers," *Microprocessors and Microsystems*, 1994.

[8] Y. Shimokawa, T. Matsushita, H. Furuno, and Y. Shimanuki, "A high-performance VLSI chip for instrumentation and electric control," *Proc. of IECON 91*, pp. 884-889, 1991.

[9] R.W. Lewis, *Programming Industrial Control Systems using IEC 1131-3.*

[10] *POSFA PLC Programming manual*, 포스콘, 1995

[11] *MCPU Programming manual*, LG 산전, 1996

장 래 혁

1989년 서울대 제어계측공학과 졸업. 동대학원 석사(1992), 동대학 박사(1996). 미시간 대학교 Research Fellow (1997). 1998년-현재 서울대학교 컴퓨터공학과 전임강사. 관심분야는 디지털 시스템 설계, 실시간 시스템, 이산현상시스템, PLC설계; 병렬 및 분산 시스템.