NEAR EAST UNIVERSITY

ELECTRICAL AND ELECTRONIC ENGINEERING

EE 400

GRADUATION PROJECT

PROGRAMMABLE LOGIC CONTROLLERS

SUPERVISOR : Özgür ÖZERDEM

PREPARED BY : Yücel YILMAZ (961108)

June – 1999

INDEX

PROGRAMMABLE LOGIC CONTROLLERS (PLC s)

1 : INTRODUCTION

In the late 1960s the American motor car manufacturer General motors was interested in the application of computers to replace the relay sequancing used in the control of its automated car plants.

Two independent companies, Bedford Associates and Allen Bradley , responded to General Motors' specification.

The computer itself, called the central processor, was designed to live in an industiial enviironment, and was connected to the outside world via racks into which input or output cards could be plugged.

The need for low-cost versatile and easily commissioned controllers has resulted in the development of *Programmable-Control systems*-standard units based on a hardware CPU and memory for the control of machines or processes. Originally designed as a replacement for the hard-wired relay and timer logic to be found in traditional control panels, PLCs provide ease and flexibility of control based on programming and executing simple logic instructions .PLCs have internal functions such as timers, counters and shift registers, making sophisticated control possible using even the smallest PLC.

A programmable controller operates by examining the input signals from a process and carrying out logic instructions on these input signals, producing output signals to drive process aquipment or machinery. Standard interfaces built in to PLCs allow them to be directly connected to process actuators and transducers whithout the need for intermediate circuitry or relays.

Through using PLCs it became possible to modify a control system without having to disconnect or re-route a single wire ; it was necessary to change only the control program using a keypad or VDU terminal. Programmable controllers also require shorter installation and commisioning times than do hardwired systems. Alhough PLCs are similar to 'conventional' computers in terms of hardware technology, they have specific features suited to industrial control :

- rugged, noise immune equipment ;
- modular plug-in construction, allowing easy replacement/addition of units
- standard input/output connections and signal levels;
- easily understood programming language
- ease of programming and reprogramming in-plant.

1.1 : Terminology (PC or PLC)

There are several different terms used to describe programmable controllers, most referring to the functional operation of the machine question :

PC programmable controller (UK origin)

PLC programmable logic controller (American origin)

PBS programmable binary system (Swedish origin)

By their nature these terms tend to describe controllers that normally work in a binary (on/off) enviroment.Since all but the smallest programmable controllers can now equipped to process analog inputs and outputs these ' labels' are not representative of their capabilities. For this reason the overall term programmable controller has been widely adopted to describe the family of freely programmable controllers. However, to avoid confusion with the personal computer 'PC', this text uses the abbreviation PLC for programmable logic controller.

1.2 : Comparison With Other Control Systems

This is only an approximate guide to their capabilities, and further technical information can be obtained from the manufacturers data sheets on each specific system.

Programmable controllers emerge from the comparison as the best overall choice for a control system, unless the ultimate in operating speed or resistance to electrical noise is required, in which case hardwired digital logic relays are chosen respectively. For handling complex functions a conventional computer is still marginally superior to a large PLC equipped with relevant function cards, but only in terms of creating the functions, not using them. Here the PLC is more efficient through passing values to the special function module, which then handles the control function independently of the main processor-a multiprocessor system.

Programmable controllers have both hardware and software features that make them attractive as controllers of a wide range of industrial equipment.

1.3 : The Advantage of (PLC) Control

Any controlsystem goes through four stages from conception to a working plant. A PLC system brings advantages at each stage.

DESIGN; The required and the control strategies decided.With conventional systems design must be complete before construction can start. With a PLC system all that is needed is a possibly vague idea of the size of the machine and I/O requirements. The input and output cards are cheap at this stage, so a healthy spare capacity can be built in to allow for the inevitable omissions and future developments.

Next comes construction. With conventional schemes, every job is a 'one-off ' with inevitable delays and costs. A PLC system is simply bolted together from standard parts. During this time the writing of the PLC program is started .

INSTALLATION ; a tedious and expensive business as sensors, actuators, limit switches and operators controls are cabled. A distributed PLC system using serial links and pre-built and tested desks can simplify installation and bring huge cost benefits. The majority of the PLC program is written at this stage.

Finally comes commissioning, and this is where the real advantages are found. No plant ever works first time. Human nature being what it is, there will be some oversights. Changes to conventional systems are time consuming and expensive. Provided the designer of the PLC system has built in spare memory capacity, spare I/O and a few spare cores in multicore cables, most changes can be made quickly and relatively cheaply. An added bonus is that all changes are recorded in the PLC's program and commissioning modifications do not go unrecorded, as is often the case in conventional systems.

MAINTENANCE ; which starts once the plant is working and is harded over to production. All plants have faults, and most tend to spend the majority of their time in some from of failure mode. A PLC system provides a very powerful toll assisting with fault diagnosis.

A plant is also subject to many changes during its life to speed production, to ease breakdowns or because of changes in its requirements. A PLC system can be changed so easily that modifications are simple and the PLC program will automatically document the charges that have been made.

2 : PLCs – HARDWARE DESIGN

Programmable controllers are purpose-built computers consisting of three functional areas: processing, memory and input/output. Input conditions to the PLC are sensed and then stored in memory, where the PLC performs the programmed logic instructions on these input state. Output conditions are then generated to drive associated eqiipment. The action taken depends totally on the control program held in memory.

In smaller PLCs these functions are performed by individual printed circuit cards within a single compact unit, whilst larger PLCs are constructed on a modular basis with function modules slotted into the backplane connectors of the mounting rack. This allows simple expansion of the system when necessary. In both these cases the individual circuit boards are easily removed and replaced, facilitating rapid repair of the system should faults development.

2.1 : Central Processing Unit (CPU)

The CPU controls supervises all operations within the PLC, carrying out programmed instructions stored in the memory. An internal communications highway, or bus system, carries information to and from the CPU, memory and I/O units, under control of the CPU. The CPU is supplied with a clock frequency by an external quartz crystal or RC oscillator, typically between 1 and 8 megahertz depending on the microprocessor used and the area of application. The clock determines the operating speed of the PLC and provides timing/synchronization for all elements in the system.

Virtually all modern programmable controllers are microprocessors-based, using a 'micro' as the system CPU. Some larger PLCs also employ additional microprocessors to control complex, time-consuming functions such as mathematical processing, three-term PID control, etc.

2.1.1 : REGISTERS

Most CPU operations involve the use of a register, which is a memory element used to store a group of bits on a temporary basis. CPU registers are located inside the microprocessor. So-called data registers are located in RAM and are used for storing flags, counter and timer constants and other types of data.A 4-bit register storesa nibble, which is 4 bits of data. An 8-bit register stores abyte, which is 8 bits of data.A 16-bit register stores aword, which is 16 bits of data.

2.1.2 : FLAG REGİSTERS

If a bit state (0 or 1) is used to indicate that some condition has ocurred it is called a flag. A register which stores a group of flag bits is called a flag register. The CPU has an internal flag register which contains information about the result of the latest arithmetical and logical operations. PLC image memory is effectively aflag register, as it contains the current status of the inputs and outputs.

2.1.3 : AUXILIARY RELAYS

Auxiliary relays are single-bit memory elements located in RAM that may be manipulated by the user's program. They are called auxiliary relays because they may be likened to imaginary internal relays. A battery-backed auxiliary relay is called a retentive or holding relays and can be used for storing data during power failure. A number of auxiliary relays may be grouped together to form a register.

It is important to remember that because auxiliary relays are only bit values stored in memory output loads cannot be connected directly to them. However, auxiliary relays can be used to control output loads indirectly.

2.1.4 : TIMERS

A CPU will have a built-in clock oscillator which controls the rate at which it operates. The CPU uses the clock signal to generate delay times.A delay times.A delay time could be used, for example, to keep an output relay energized for a fixed period.

2.1.5 : SHIFT REGISTER

Some register are arranged so that bits stored in them can be moved one position to the left or to the right with the application of a shift command or pulse.Such registers are called shift registers and can be used for sequence control applications

2.1.6 : BINARY COUNTER

The CPU may functions as a binary counter since it isable to increment and decrement binary data stored in a register and compare binary data stored in two seperateregisters.Counters are used to count, for example,digital pulses generated from a switching device connected to an input port. An output is usually generated after a predetermined number of input pulses have been counted. The count value required is stored in a data register.

2.2 : Memory

Memory is charecterized by its volatility. A memory is volatile if it loses its data when the power to it is switched off and non-volatile otherwise. Common types of memory include semiconductor memory and magnetic disk. The various types of semiconductor memory are :

1. *RAM* Random access memory is a flexible type of read/write memory. All PLCs will have some amount of *RAM*, which is used to store ladder programs being developed by the user, program data which needs to be modified and image data.

   RAM is volatile. This means that *RAM* cannot be used to store data while the PLC is turned off unless the RAM is battery backed. A type of *RAM* called *CMOS RAM* (complementary metal-oxide semiconductor *RAM*) is suitable for use with batteries because it consumers very little power and operates over a very wide range of supply voltages.

2. *ROM* A read only memory is programmed during its manufacture using a mask. It is a non-volatile memory and provides permanent storage for the operating system and fixed data.

3. *EPROM* Erasable programmable read only memory is a type of *ROM* which can be programed by electrical pulses and erased by exposing a transparent quartz window found in the top of each device to ultraviolet light. *EPROM* is non-volatile memory and provides permanent storage for ladder programs.

4. *EEPROM* Electrically erasable programmable read only memory is similar to *EPROM* but is erased by using electrical pulses rather than ultaviolet light. It has the the flexibility of battery-backed *CMOS RAM*. However, writing data into an *EEPROM* takes much longer than into a *RAM*.

2.2.1 : MEMORY STORAGE CAPACITY

The storage capacity of a memory device is determined by the number of binary digits,i.e. on/off states, it can hold. In microelectronics, 1 K represents the number 1024 i.e. the binary number 2 . A 4K byte memory is capable of 4*1024 words, each of 8 bits, and has a total storage capacity of 32 768 bits.

Clearly, the storage capacity of the user memory will determine the maximum program size. As a guide, a 1K byte memory will hold 1024 program instructions and data if these are stored as groups of 8 bits.

2.2.2 : MEMORY MAP

We use the term memory mapping to describe the situation in which input/output ports are controlled by writing data into the image memory. A diagram which shows the allocation of memory addresses of ROM, RAM and I/O is called a memory map.In this image bits are stored in RAM above the user's program and data for flags, counters and timers. Flags,counters and timers are discussed below. With most PLCs the memory map is already configured by the manufacturer. This means that the program capacity, the number of input/output ports and the number of internal flags, counters and timers are fixed.

2.3 : Multitasking

More advanced PLCs use multitasking. This is the process of running two or more control tasks using a single CPU. Each tasks has its own program and allocated input/output ports. The CPU may schedule its prcessing time among the various tasks or allow events to initiate the various tasks. Tasks are assigned priority levels.Higher-priority tasks are always executed before lower-priority tasks.

Multitasking systems make use of interrupts. An interrupt is a special control signal to the CPU which tells it to stop executing the program in hand and start executing another program stored elsewhere in memory. The CPU clock oscillator can be used to provide interrupts at regular intervals so that processor time can be shared between tasks. Alternatively, an external event such as a machine fault alarm can be used to drive the interrupt line.

2.4 : Types of ports (input/output units)

Most PLCs operate internally at between 5 and 15 V d.c (common TTL and CMOS voltage), whilst process signals can be much greater, typically 24 V d.c. to 240 V a.c. at several amperes.

The I/O units form the interface between the microelectronics of the programmable controller and the real world outside, and must therefore provide all necessary signal conditioning and transducers whithout the need for intermediate circuitry or relays.

To provide this signal conversion programmable controllers are available with a choice of input/output units to suit different requirements. For example;

Inputs	(choice of):	5 V	(TTL level) switched I/P
		24 V	swiched I/P
		110 V	swıched I/P
		240 V	switched I/P
Outputs	(choice of):	24 V	100 mA switched O/P
		110 V	1 amp
		240 V	1 A a.c. (triac)
		240 V	2 A a.c. (relay)

It is standard practice for all I/O channels to be electrically isolated from the controlled process, using opto-isolator circuits on the I/O modules. An opto-isolator circuit consists of a light-emitting diode and a photo-transistor, forming an opto-coupled pair that allows small signals to pass through, but will clamp any high-voltage spikes or surges down to the same small level. This provides protection against switching transients and power-supply surges, normally up to 1500 V.

In small self-contained PLCs in which all I/O points are physically located on the one casing, all inputs will be of one type and the same for outputs. This is because manufactererssupply only standard function boards for economic reasons. Modular PLCs have greater flexibility of I/O however,since the user can select from several different types and combination of input and output modules.

In all cases the input/output units are designed with the aim of simplifying the connection of process translucers and actuators to the programmable controller.

For this purpose all PLCs are equipped with standard screw terminals or plugs on every I/O point, allowing the rapid and simple removal and replacement of a faulty I/O card.

Every input/output point has a unique address or chanel number which is used during program development to specify the monitoring of an input or the activating of a particular output within the program. Indication of the status of input/output chanels is provided by light-emitting diodes on the PLC or I/O unit, making it simple to check the operation of process inputs and outputs from the PLC itself .

2.4.1 : ANALOGUE PORTS :

Many types of transducer produce analogue signals variable-speed motor drives are controlled by an analogue speed command signal.consequently, PLC manufacturers provide ports for handling analogue signals as well as digital. This are based on analogue to digital converters (ADCs) and digital to analogue converters (DACs).

2.4.2 : COMMUNICATIONS PORTS :

Many PLCs have ports for network communications and for interfacing to a computer.

1. Presenting operating data and alarm, etc.via printers or VDUs.
2. Data logging into archive files or record : to be used for process performance analysis and management information.
3. Passing values/parameters into existing PLC programs from operator terminals or supervisory controllers.
4. Forcing I/O points and memory elements from a remote terminal.
5. Changing resident PLC programs-uploading/from a supervisory controller
6. Linking a PLC into a control hierarchy containing several sizes of PLC and computer.

2.5 : Power Supplies

The CPU, memory and input/output are electronic companents which require power (typically +5 V d.c. and +/- 15 V d.c. at a few milliamperes). A PLC incorporates a power supply for powering internal companents and input ports.

Power supplies fall into two categories : linear and swich mode. A linear power supply uses a simple regulator circuit to convert the mains supply to a constant d.c. voltage. A swich-mode power supply uses a high-frequancy swiching regulator to produce a series of pulses. Averaging the pulses provides a smooth d.c. voltage. The main advantages of a swich-mode power supply are : (a) it is capable of providing a vide range of supply voltage. (e.g. +/- 24 V d.c., +/- 15V d.c., +/- 5 V d.c., 0 V), (b) swich action makes it highly efficiend so that the amount of heat dissipated from the supply is small, and (c) it is compact and lightweight. Becomes of these advantages the swich-mode power supply is often used in PLCs .

3 : TYPES OF PLC SYSTEM

The increasing demand from industry for programmable controllers that can be applied to different forms and sizes of control tasks has resulted in most manufacturers producing a range of PLCs with various levels of performance and facilities.

Typical rough definitions of PLC size are given in terms of program memory size and the maximum number of input/output points the system can support.

However ,to evaluate properly any programmable controller we must consider many additional features such as it processors , cycle time, language facilities , function , expansion capability ,etc...

A brief outline of the characteristics of small ,medium and large programmable controllers is given below , together with typical applications.

PC size	Max I/O Points	User memory size (no.of instructions)
Small	40/40	1 K
Medium	128/128	4 K
Large	>128/>128	> 4 K

3.1 : Small PLCs

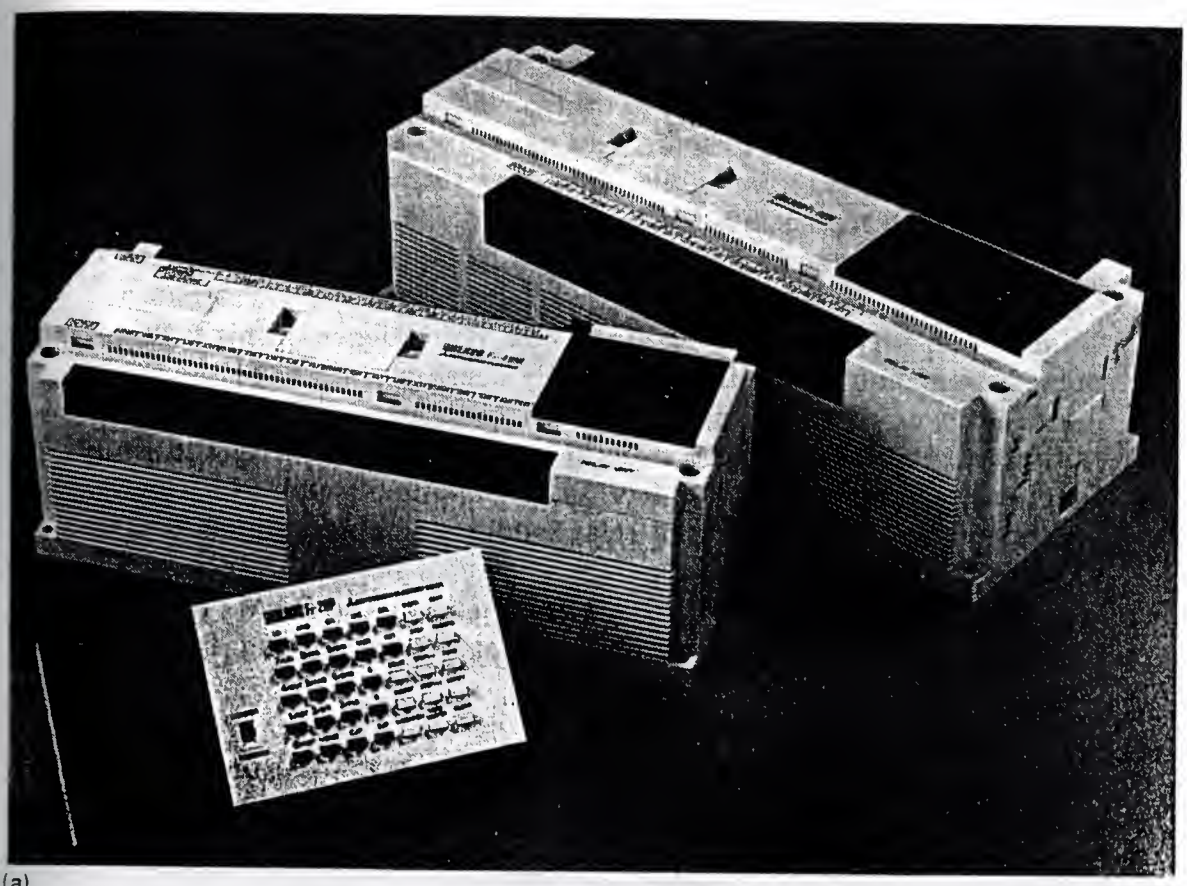
In general, small and 'mini' PLCs are designed as robust ,compact units which can be mounted on or beside the equipment to be controlled . They are mainly used to replace hard-wired logic relays , timers , counters , etc.. That control individual items of plant or machinery , but can also be used to coordinate several machines working conjunction with each other.

Small programmable controllers can normally have their total I/O expandet by adding one or two I/O modules , but if any further developments are required this will often mean replacement of the complete unit.

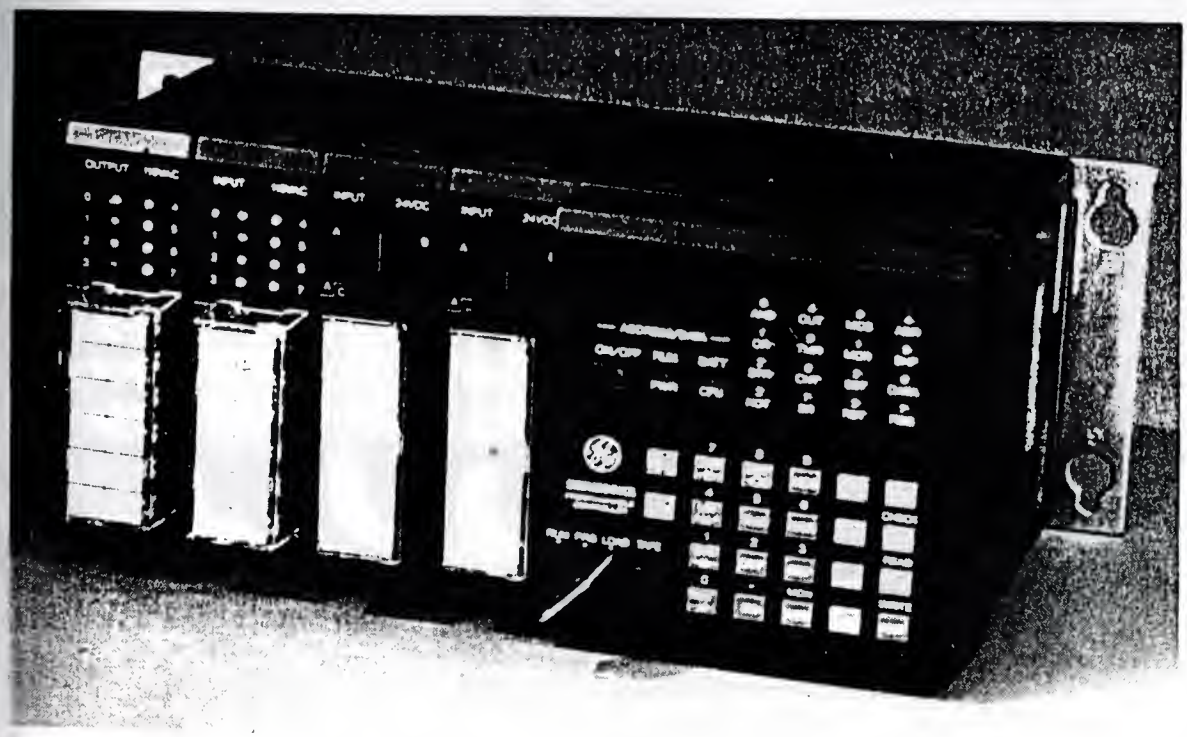
This end of the market is very much concerned with non-specialist end-users , therefore ease of programming and a 'familiar' circuit format are desirable. Competition between manufacturers is extremely fierce in this field, as they vie to obtain a maximum share in this partially developed sector of the market.

A single processor is normally used, and programming facilities are kept at a fairly basic level, including conventional sequencing controls and simple standard functions : e.g. timers and counters. Programming of small PLCs is by way of logic instruction lists or relay ladder diagrams.

Program storage is by EPROM or battery-backed RAM. There is now a trend towards EEPROM memory with on-board programming facilities on several controllers.

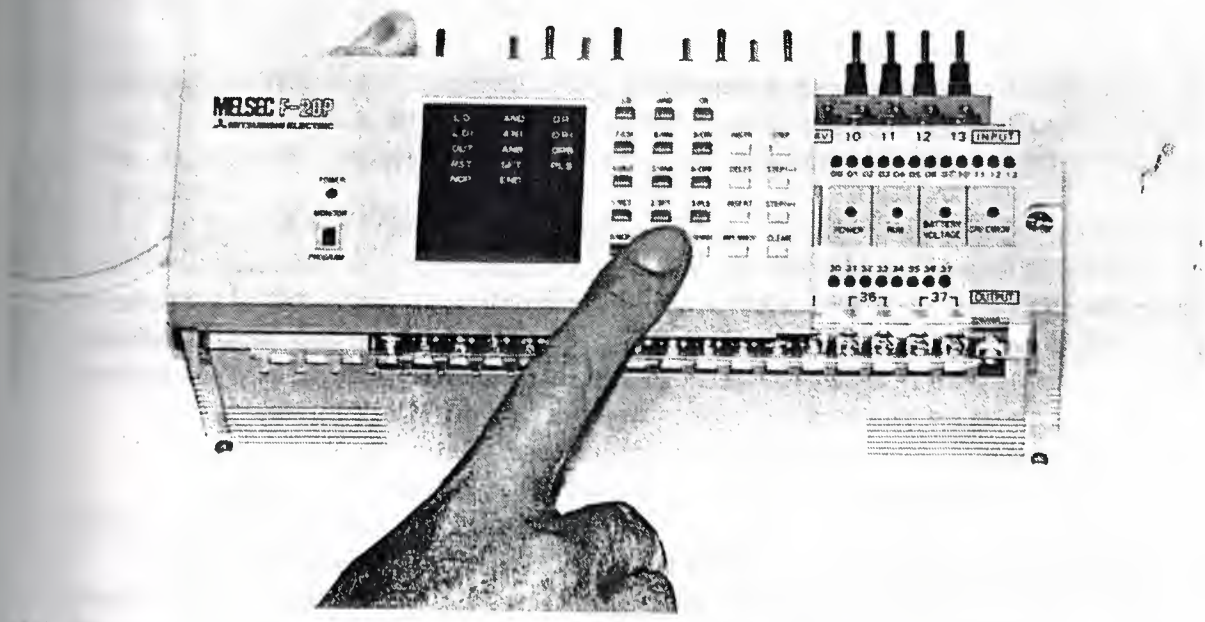


(a)

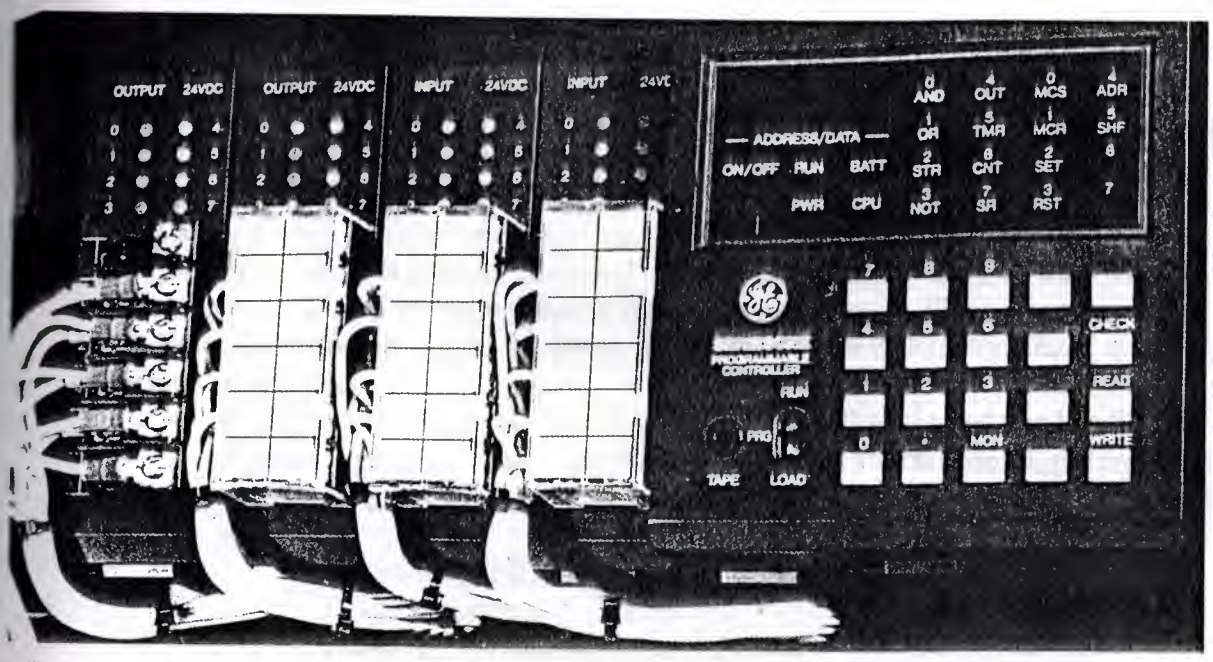


(b)

Small PLCs: (a) Mitsubishi F40 (courtesy Mitsubishi Electric UK Ltd); (b) GE series 1 (courtesy General Electric).



(a)



(b)

Small PLCs: (a) Mitsubishi F series (courtesy Mitsubishi Electric UK Ltd); (b) GE series 1 (courtesy General Electric).

3.2 : Medium-size PLCs

İn this range modular construction_predominates with plug-in modules based around the Eurocard 19 inch rack format or another rack mounting system. This construction allows the simple upgrading or expansion of the system by fitting additional I/O cards into the rack, since most rack systems have space for several extra functions cards. Boards are usually 'ruggdized' to allow reliable operations over a range of environment.

İn general this type of PLC is applied to logic control tasks that cannot be met by small controllers due to insufficient I/O provision , or because the control task is likely to be extended in the future. This might require the replacement of a small PLC , whereas a modular system can be expanded to a much greater extent , allowing for growth.. A medium-sized PLC may therefore be financially more attractive in the long term.

Communication facilities are likely to be provided, enabling the PLC to be included in a 'distributed control' system.

Combinations of single and mulyi-bit processors are likely within the CPU. For programming, standard instructions or ladder and logic diagrams are avaliable. Programming_ is normally carried out via a small keypad or a VDU terminal.

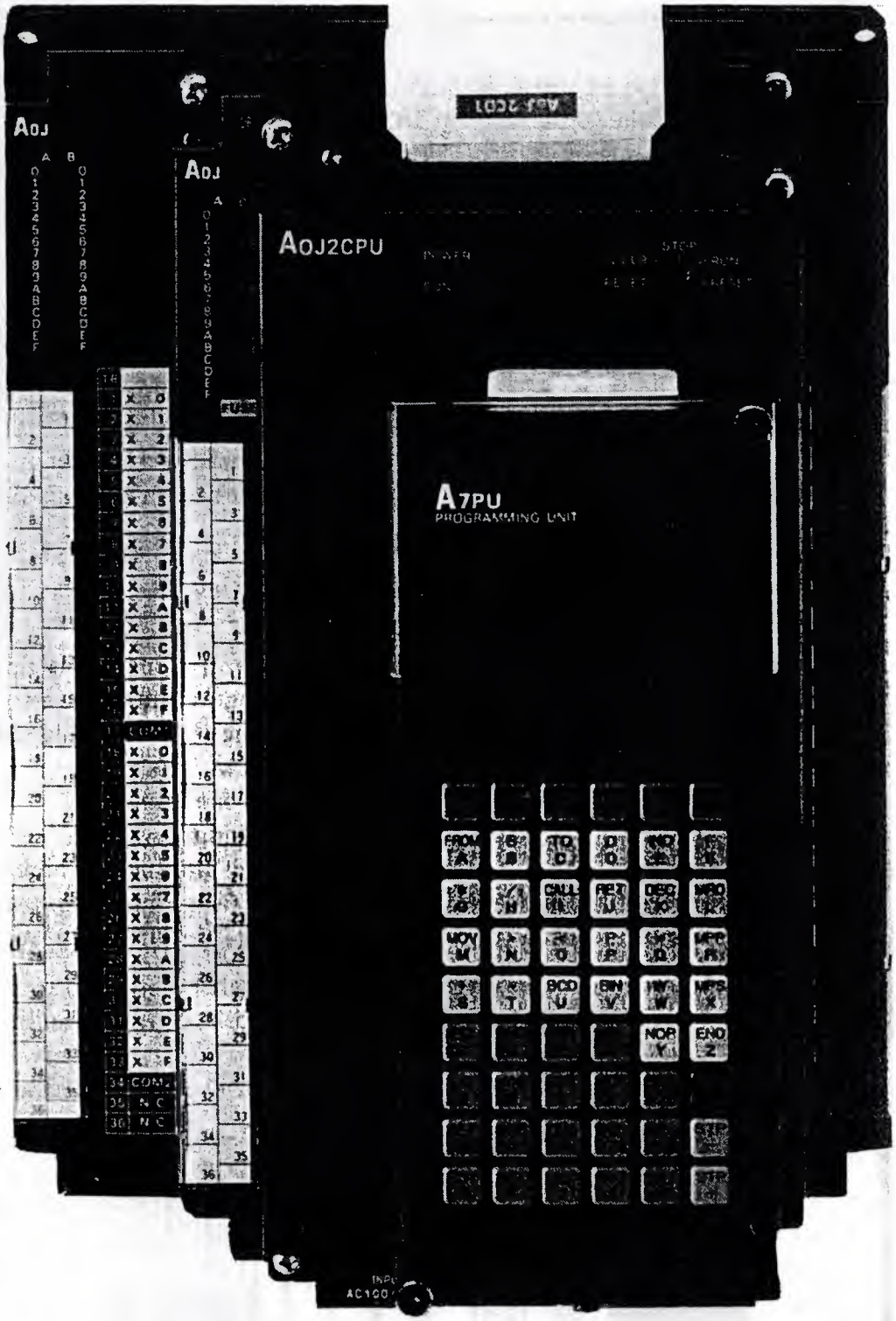
3.3 : Large PLC

Where control of very large numbers of input and output points is necessary or complex control functions are required, a large programmable controller is the obvious choice. Large PLCs are designed for use in large plants or on large machines requiring continuous control. They are also employed as supervisory controllers to monitor and control several other PLCs or intelligent machines, e.g. CNC tools.

Modular construction in Eurocard format is standard, with a wide range of function cards available including analog input/output modules. There is a move toward 16-bit processors, and also multi-processor usage in order to efficiently handle a large of differing control tasks.

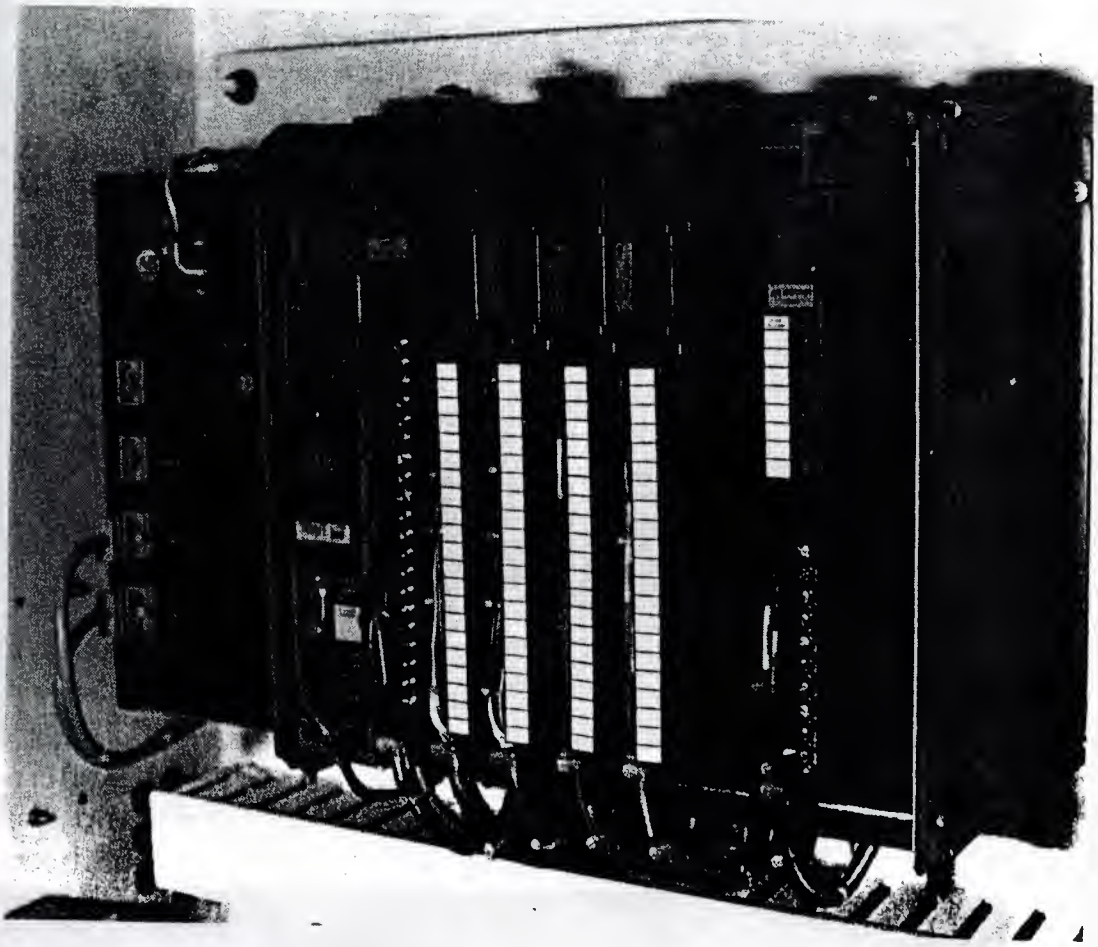
- 16-bit processor as main processor for digital arithmetic and text handling.
- Single-bit processors as co-or parallel processors for fast counting,storage,etc.
- Peripheral processors for handling additional tasks which are time-critical, such as:

  Cosed-loop (PID) control
  Position controls
  Floating-point numerical calculations
  Diagnostics and monitoring
  Communications for decentralized I/O
  Process mimics (screen graphics)
  Remote input/output racks.

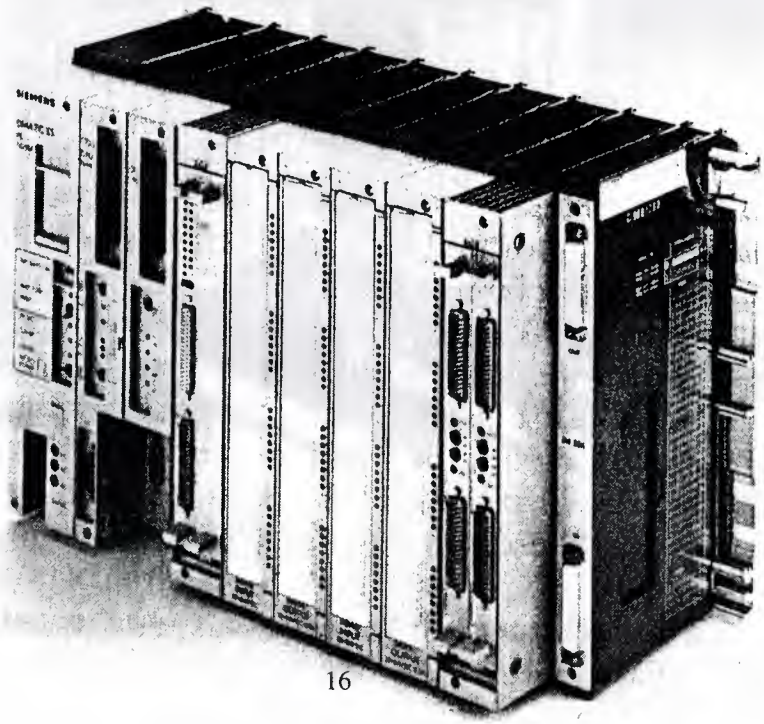


Medium-sized PLCs: Mitsubishi AO PLC

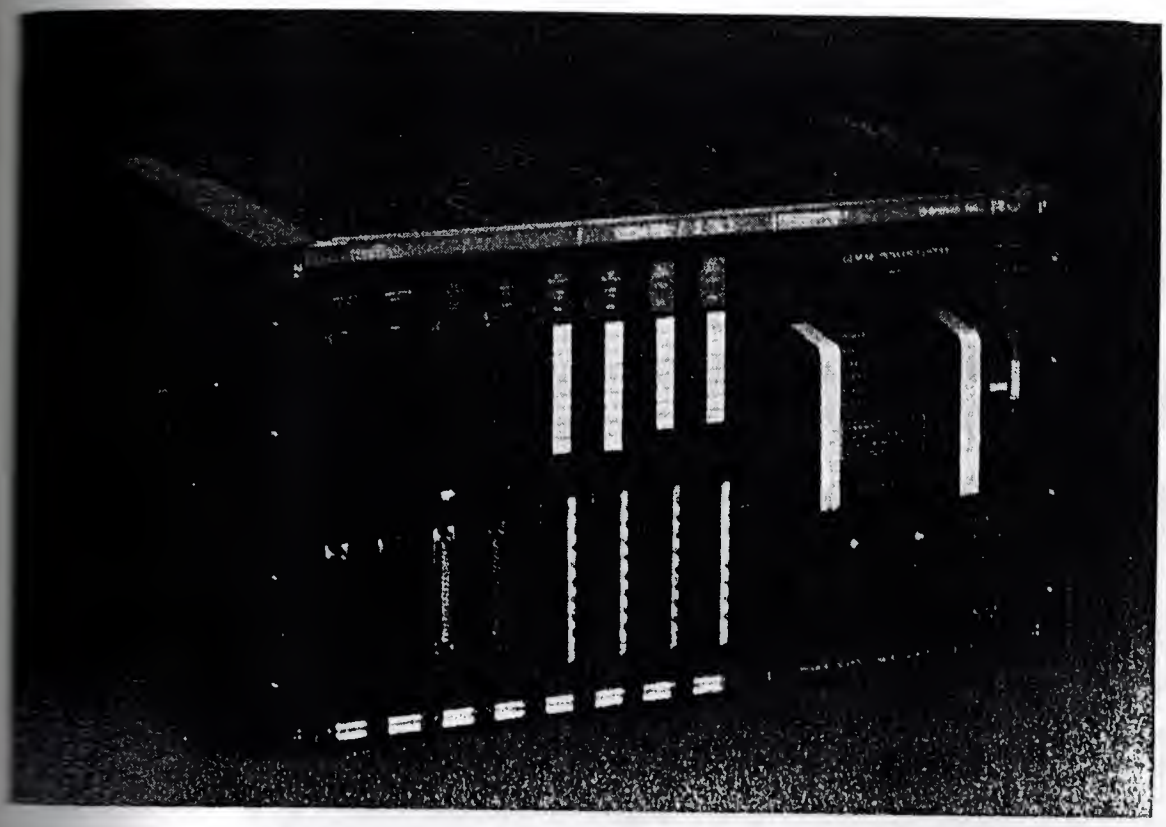
The four medium-sized PLCs discussed:



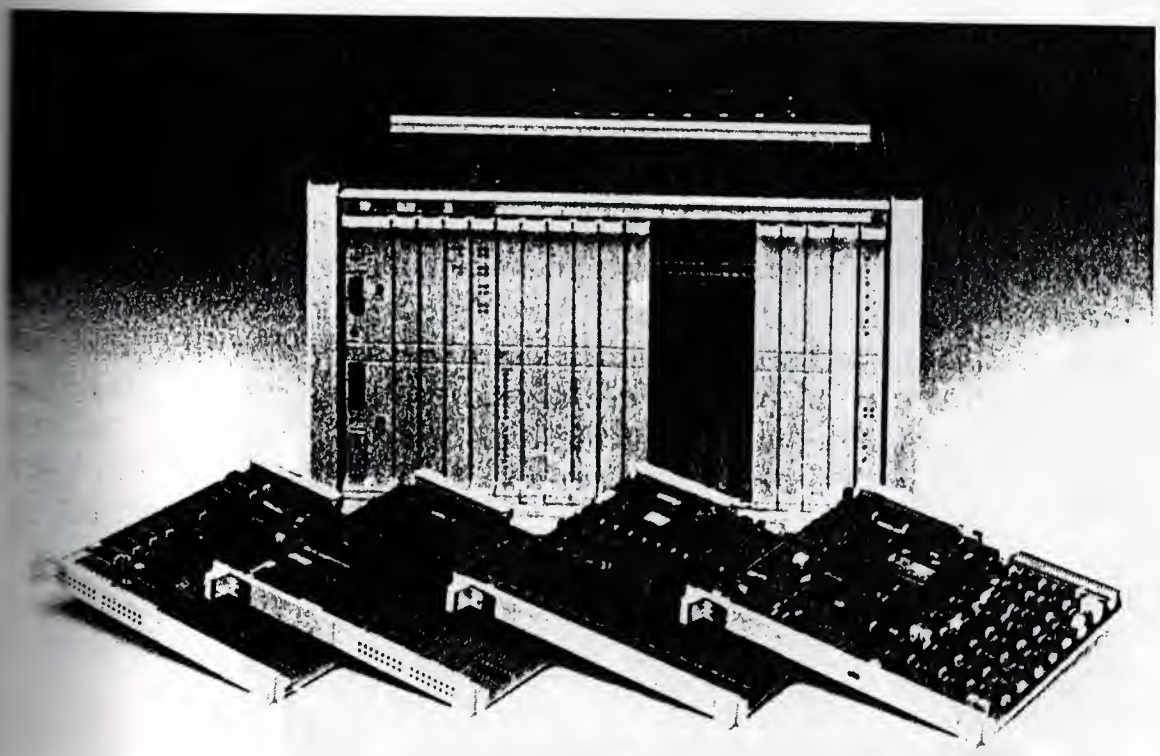
(a) the Allen Bradley PLC-5;



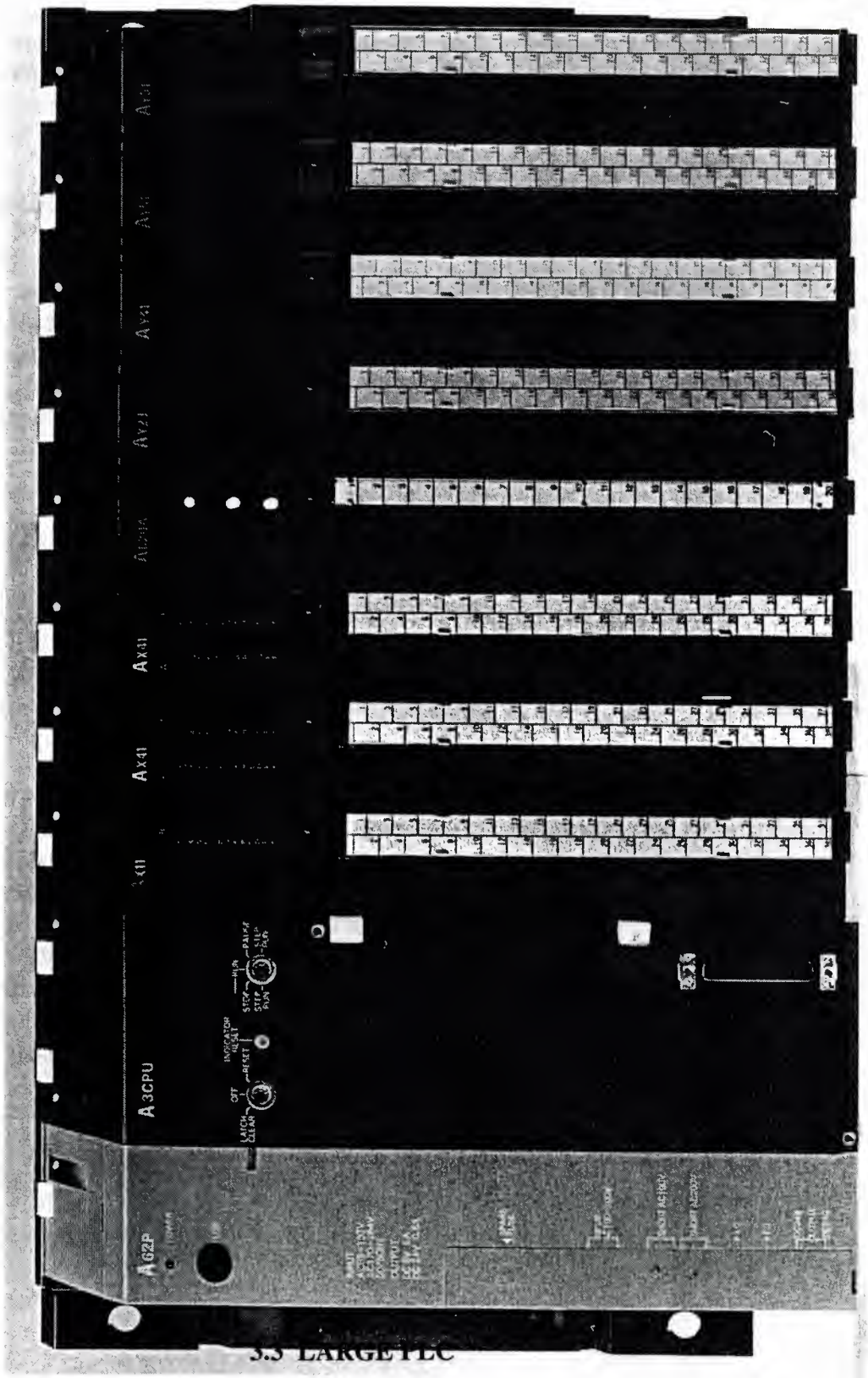
(b) the Siemens S5–1154;



(c) the CEGELEC GEM-80;



(d) the ABB Master. Photographs courtesy of the manufacturers



3.3 LARGE PLC

This multi-processor solution optimizes the performance of the overall system as regards versatility and processing speed, allowing the PLC to handle very large programs of 100K instructions or more. Memory cards can now provide several megabytes of CMOS RAM or EPROM storage.

3.4 : Remote input/output

When large numbers of input/output points are located a considerable distance away from the programmable controller, it is uneconomic to run connecting cables to every point. A solution to this problem is to site a remote I/O unit near to the desired I/O points. This acts as a concentrator to monitor all inputs and transmit their status over a single serial communications link to the programmable controller. Once output signals have been produced by the PLC they are fed back along the communications cable to the remote I/O unit, which converts the serial data into the individual output signals to drive the process.

4 : PLCs- SOFTWARE ENGINEERING

Figure 4.1 shows the six stages that any software project must go through during its life. Although few projects are compartmentalized as neatly as this, the principles apply to all.

The first stage is analysis of the problem that is to be solved. The supplier / programmer of the PLC system must meet with the other contractors and the user to determine what controls are needed and how the control actions are to be provided. Important considerations such as operator controls need to be established at this stage .Ambiguous descriptions should be resolved.

Of all the stages, analysis is the most difficult, as the ultimate end-user and the other contractors probably have not considered the intricacy of the control stategy, and do not have the experience to decide if an item of plant is best controlled with joysticks, pushbuttons or a touchscreen VDU.

An important point which is often overlooked at this stage is the need to provide some form The output from the analysis stage should be a description of how the plant work, what operator stations and controls are needed, what maintenance/fault-finding aids and facilities are tobe included and finally a complete list of the I/O signals with voltage /current specifications and their locations on the plant.
of manual 'maintenance' controls to test, or rescue, afully automated plant or sequence which has failed in some obscure manner.

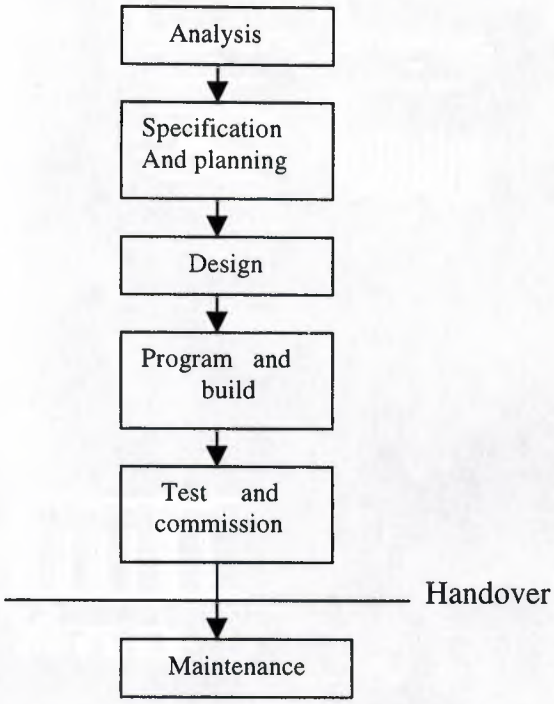


Figure 4.1 The stages of a project

The difficulties of this first stage cannot be overemphasized. If the ambiguities and problems are resolved at the start, the following stages are easy. Finding out at the commisioning stage that the user wanted variable speed fans and an underpressure alarm and 'thought you knew that' is not the way to ensure a smooth plant start-up. If in doubt ask; even if you are not in doubt, still ask, and assume nothing.

At this stage, the final testing requirements should also be defined. If you do not know how you are going to test it, how will you know if the plant meets the user's requirements.

Wh the worst stage over, the designer should produce a description of what the control system contains, how it is going to perform and how it will be tested.This is really recording what was agreed at stage 1.

The next stage is to design the system ; the cubicles, desk, and the stracture of the program. This latter action, known as top-down design.

At the last the programming can be done, built around the structure laid down at the design stage. No program should be constructed *adhoc* at the keyboard; that way lies spaghetti programming. Commercial programmers estimate that this stage generally involves no more than 10% of the total effort.

With the programming completede and the plant built, testing and commisionning can start. The operation should be checked against the spesificationsproduced stage two. With all bar the simplest system, it can be very time consuming to check all routes and actions given in the specifications. There is generally pressure to 'handover' the plant when the basic operation has been tested but the ancillary, rarely used, options are untried.Too often these tests are skipped, and the first time a 'firkling fault' mode is tested is when the 'firkling fault' first occurs, possibly years after the plant has started up.Inevitably, commissioning of the control system will always be the last stage in a new plant, so the control engineer ends up carrying everyone elses delay. It is therefore important to establish what testing must be carried out before a plant can start and what can be tested lated later, on line. On line testing, however, can be very difficult and time consuming.

Safity-related checks should never be skipped; finding out that an emergency stop sequence does not work when it is used for the first time Health and Safety Executive.

The final stage is usually overlooked. Once the plant is handed over,its control ststem must be maintained, a term used here not to mean serviced in the mechanical sense but covering fault finding, resolving of bugs and changes arising from modifications in the way the plant operates. No plant is fixed, all change during their life in response to market or technology changes, and these modifications require changes in the control strategy.

In commercial programming it is generall thought that maintenancetakes over 50% of the effort in a project's life cycle. It is therefore essential that the control strategy and program are constructed and documented so they can be changed and modified easily at a later stage, possibly by people who had no involvement whith the previous five stages.

4.1: PLC Operating System

In all PLC operating systems similar operating systems are used. These programs are in ROM and they are loaded into the system while manifucturing.

In general a PLC operating system does the following :

- Operates the user program.
- Event and time dependend service programs are operated by operation system.
- Orginize the communication of PLC and controls the operation of the system.

4.2 :General Physical Build Mechanism

PLCs are seperated in to two according to their bulding mechanism

- ***Compact PLCs :*** Are manufactured such that all units forming the PLC are placed in a casp they are low price PLC with lower capacity they are usually prepared by small or medium size machine manufactures. In some types compact enlargement module is present.

e.g. Siemens S7 –200 , Omron SK-20

- ***Model PLCs :*** They are formed by combining separate modules (called RACK) together in a board. They can have different memory capacitiy, I/O numbers, power supply up to necessary units.
  e.g. Siemens S5-115U, Omron C200H

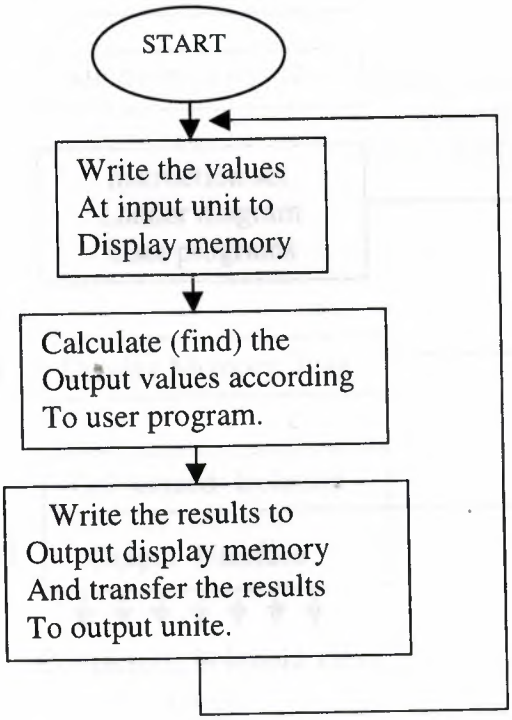
4.3: User Program Operation

A user program loaded to program memory of PLC starting from the first instruction untill the last instruction executes the instructions step by step. If there is a jump or branching in the program the instructions until the jump address are not executed. When the last instruction is reached it automatically turns to first instruction. This operation is like an infinite loop.

The time taken by the PLC to turn back to the same instruction is called the scanning period. The scanning period of a PLC is depending upon I/O number, programs length and operating frequency of the CPU.

e.g. A PLC with 500 word program capacity and with 10 input and output signal is 2.6 ms and program execution time is 12 ms. In general scanning time of PLCs differ from 2-200 ms. Scanning velocity is usually operating velocity per 1024 bytes.

4.4 : Flow Diagram for Executing the Program



In some PLCs the output data are send to output units (DSP : Direct Pocessing System) Hitachi H200.

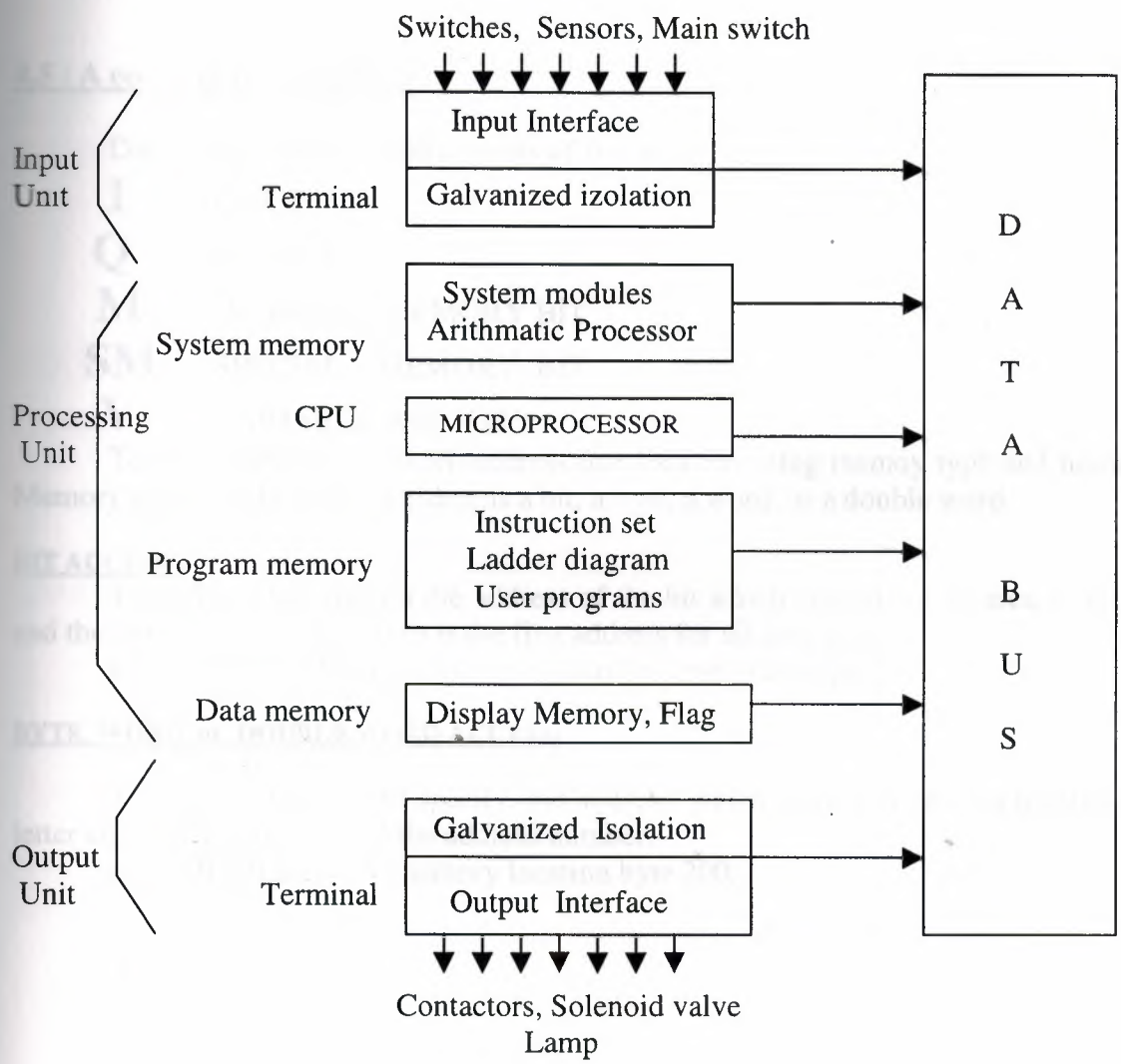
In some PLCs you can reach real input and real outputs directly by some instructions (immediate I/O instructions) Simatic S7.

4.4 :Internal Structure of PLCs

They have three main units

- Input unit
- Processing unit
- Output unit

BLOCK DIAGRAM REPRESANTATION:



INPUT UNIT: Is the unıt that converts the signal coming from the control elements of the system in to logic levels. The analog and/or digital signals coming from the sensors switches showing the system pressure, humidity etc. enters the PLC throuhg the input unit.Digital input signals are usually 24V dc or according to the medium can be 48V dc, 110V dc or 240V dc.

Analog signals are standart 0.....10V , -5V.....0.....+5V, -10V.....0.....+10V or 0/4.....20 mA.

Digital signals are converted to 5V dc by this unıt which is the internal voltage level of the device Analog signal on the other hand according to the type of ADC are converted to 8,12,14 or 16 bit numerical valve.

The parasitic signals are first filtered by RC passive filter and then they pass through optocoupler that has the property to supply galvinized isolation. As the result of this process the signals are send to input display memory. Analog signals pacs through frequency in some PLC. In this way they gain important noise immunity.

4.5 : A ccessing Data Memory

Data memory for S7-200 consısts of five areas.

I INPUT

Q OUTPUT

M INTERNAL MEMORY BİT

SM SPECIAL MEMORY BIT

V VARIABLE MEMORY

To use a memory locatıon, address that location using memoy type and number. Memory areas can be accesed either as a bit, a byte, a word, or a double word.

BIT ACCESS :

To access a bit, specify the address of the bit which consists of an area identifier and the byte or bit number. Zero is the first address for all data areas.

e.g : I 0.0 Bits address is a decimal number from 0 through 7.

BYTE, WORD or DOUBLE WORD ACCESS:

To access a byte, word specify, the address, which consist of an area identifier, a letter signifying data size and the address number.

e.g : VB200 access V memory locatıon byte 200.

5 : PROGRAMMING TECHNIQUES

There are three programming techniques for PLCs.

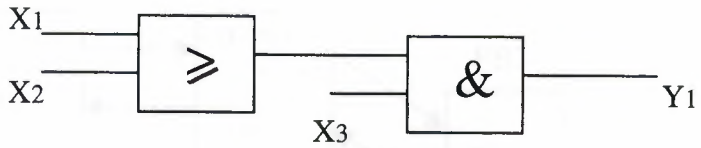
- Statement list or instruction list programming.
- Ladder programming.
- Other programming techniques (Logic gates , symbolic)

EXAMPLE :

LOGIC EXPRESSION: All the programs above are about the logic expression

$$Y_1 = (X_1 + X_2) * X_3$$

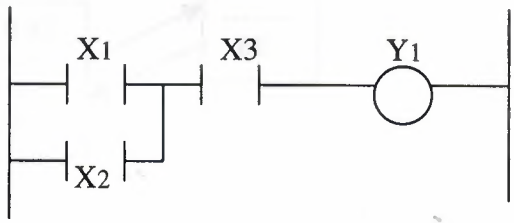
LOGIC GATES: First two techniques are for hand programmers but with PCs it is possible to use all.



STATEMENT LIST:

LD	X1
OR	X2
AND	X3
OUT	Y1

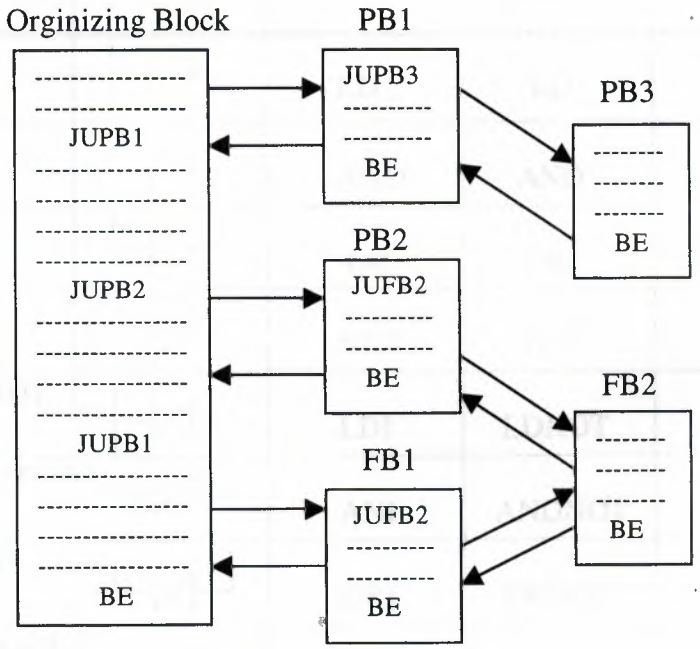
LADDER PROGRAMMING:



Programming methods are divided into two (two main; groups according to the way they are writen)

1. ***Step by step programming:*** In step by step programming instructions are written one after the other and they executed in the same way. In one cycle all instructions are executed and all instructions are in the main program.
2. ***Structure programming:*** Programs are written in blocks and by the help of orginizing block the other blocks that are going to be executed in one cycle are executed.

It is not necessary for all the instructions in a structure programming to be executed. Depennding upon the instructions in organizing block some blocks cannot be executed. The data for the blocks that are not executed are kept in the memory.



Structure programming

5.1 : Programming:

Usually basic logic instructions are enough construct a control panel and if timer instructions are added to these basic logic instructions then it is very easy to construct any contactor panel.

The instructions necessary to implement a logic function with PLC can be divided into three groups.

GROUP 1 : Starting instruction like *LOAD , LOAD NOT*

GROUP 2 : Basic logic function instruction,like, *AND ,OR , NOT , AND NOT , OR NOT OR NOT* , end of function instruction like , *AND BLOCK , OR BLOCK*

GROUP 3 : The output assigment instructions , *OUT*

INSTRUCTION	LADDER SYM	HITACHI	OMRON	MITSUMI	TEXAS INST	SIMATIC S7
LOAD		LD	LD	LD	STR	LD
AND		AND	AND	AND	AND	A
OR		OR	OR	OR	OR	O
NOT		NOT	NOT	I	NOT	NOT
LOAD NOT		LDI	LDNOT	LDI	STRNOT	LDN
AND NOT		ANI	ANDNOT	ANI	ANDNOT	AN
OR NOT		ORI	ORNOT	ORI	ORNOT	ON
AND BLOCK		ANB	ANLD	ANB	ANDSTR	ALD
OR BLOCK		ORB	ORLD	ORB	ORSTR	OLD
OUT		OUT	OUT	OUT	OUT	=
END		END	END	END	END	MEND

In additional to these *TIMER* , *COUNTER* and *CONTROL STATEMENTs* are available.

In all PLCs basic logic functions does the job and they are programmed similary. There may be some difference in timer, counter and control statement.

An assumption is going to be made while considering PLC programs. The assumption is that while PLC programs are used to execute the logic functions, accumulation memory is used and the top level of the memory is going to be assumed as accumulation.

5.1.1 :LOGIC FUNCTION START INSTRUCTION

When these instructions are executed the data is load to accumulator or to the first level of the memory.

In the following example the execution of the load instruction in a four accumulation memory is shown.

LOAD X1

(After execution of inst. X1 is loaded the first level of the acc. Memory and the rest of the data is shifted below.)

BEFORE	AFTER
D0	**X1**
D1	**D0**
D2	**D1**
D3	**D2**

LOAD NOT X1

(Is not applicaple to SIMATIC S5PLC)

BEFORE	AFTER
D0	**X1'**
D1	**D0**
D2	**D1**
D3	**D2**
D4	**D3**

5.1.2 : BASIC LOGIC FUNCTION INSTRUCTIONS

For these instructions the logic function stated by the instruction is performed by the data given by the instruction and the data at the first level of the memory.
(*AND* , *OR* , *AND NOT* ,*OR NOT*)

AND X1

BEFORE	AFTER
D0	**X1.D0**
D1	**D1**
D2	**D2**
D3	**D3**

OR X1

BEFORE	AFTER
D0	**X1+D0**
D1	**D1**
D2	**D2**
D3	**D3**

5.1.3 : END OF FUNCTION STATEMENTS

When statement is executed the data at (first level of memory) and the data in the second level are used to execute the statement. And the result is written to the first level.

AND BLOCK

BEFORE	AFTER
D0	**D0.D1**
D1	**D2**
D2	**D3**
D3	**-------**

OR BLOCK

BEFORE	AFTER
D0	**D0+D1**
D1	**D2**
D2	**D3**
D3	**-------**

5.1.4 : ASSIGMENT TO OUTPUT STATEMENT

When executed data at accumulator is send to output ***OUT Y1.***

EXAMPLE :

TEXAS INST.		SIMATIC S7	
STR	X9	LD	X9
OR	C1	O	C1
STR NOT	X10	LDN	X10
AND	X11	A	X11
AND	STR	A	LD
OUT	C1	=	C1
STR	X12	LD	X12
OR	C2	O	C2
STR	C3	LD	C3
AND	STR	A	LD
OUT	C4	=	C4
END		MEND	

ACC. 1st level	*ACC. 2nd level*
X9	-
C1+X9	-
X10'	*C1+X9*
X10'. X11	*C1+X9*
(X10'. X11).(C1+X9)	-
(X10'. X11).(C1+X9)	-
X12	*(X10'. X11).(C1+X9)*
C2+X12	*(X10'. X11).(C1+X9*
C3	*C2+X12*
C3.(C2+X12)	-
C3.(C2+X12)	-

Special Memory (SM) Bits

Special memory bits provide a variety of status and control functions, and also serve as a means of communicating information between the CPU and your program. Special memory bits can be used as bits, bytes, words, or double words.

SMB0: Status Bits

As described in Table D-1, SMB0 contains eight status bits that are updated by the S7-200 CPU at the end of each scan cycle.

Table D-1 Special Memory Byte SMB0 (SM0.0 to SM0.7)

SM Bits	Description
SM0.0	This bit is always on.
SM0.1	This bit is on for the first scan. One use is to call an initialization subroutine.
SM0.2	This bit is turned on for one scan if retentive data was lost. This bit may be used as either an error memory bit or as a mechanism to invoke a special start-up sequence.
SM0.3	This bit is turned on for one scan when RUN mode is entered from a power up condition. This bit may be used to provide machine warm-up time before starting an operation.
SM0.4	This bit provides a clock pulse that is on for 30 seconds and off for 30 seconds, for a cycle time of 1 minute. It provides an easy-to-use delay, or a 1-minute clock pulse.
SM0.5	This bit provides a clock pulse that is on for 0.5 seconds and then off for 0.5 seconds for a cycle time of 1 second. It provides either an easy-to-use delay or a 1-second clock pulse.
SM0.6	This bit is a scan clock which is on for one scan and then off for the next scan. This bit can be used as a scan counter input.
SM0.7	This bit reflects the position of the Mode switch (off is TERM position, and on is RUN position). If you use this bit to enable freeport mode when the switch is in the RUN position, normal communication with the programming device can be enabled by switching to the TERM position.

5.2 : Input/Output Numbering

Different *PLC* manufactures use different numbering systems for input/output points and for other functions within the controller.

From now on we will use the following assigment.

Input	I0.0 ---------- I0.8	Siemens
Output	Q0.0 ---------- Q0.8	Simatic S7-200

Implementation of logic gates in *PLC*.

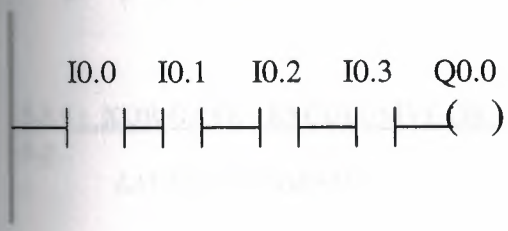
5.3 : Some Special Ladder Instructions With Examples:

5.3.1 : AND GATE

In order to activate the output Q0.0 all contants should be activated.

e.g.:

LADDER PROGRAM.



STATEMENT LIST

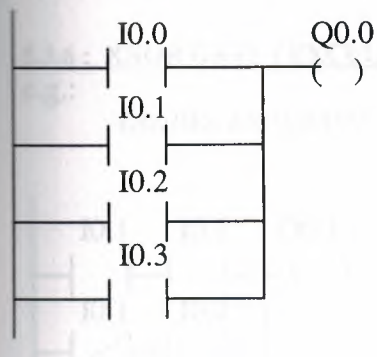
(Simatic S7)		(Texas Inst.)	
LD	I0.0	STR	I0.0
A	I0.1	AND	I0.1
A	I0.2	AND	I0.2
A	I0.3	AND	I0.3
=	Q0.0	OUT	Q0.0

5.3.2 : OR GATE

If any of the contants is activated then the output Q0.0 is activated.

e.g.:

LADDER PROGRAM.



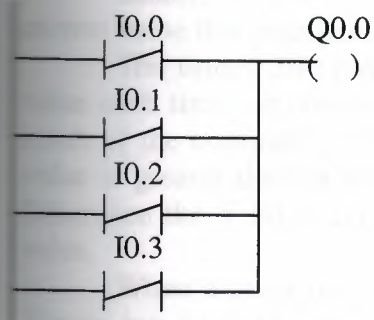
STATEMENT LIST

(Simatic S7)		(Texas Inst.)	
LD	I0.0	STR	I0.0
O	I0.1	OR	I0.1
O	I0.2	OR	I0.2
O	I0.3	OR	I0.3
=	Q0.0	OUT	Q0.0

5.3.3 : NAND GATE If all contacts are opened Q0.0 is deactivated.

e.g.:

LADDER PROGRAM.



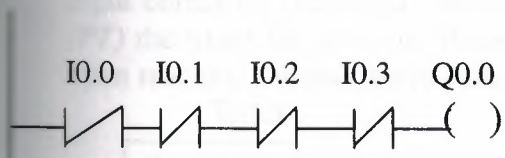
STATEMENT LIST

(Simatic S7)		(Texas Inst.)	
LDN	I0.0	STRNOT	I0.0
ON	I0.1	ORNOT	I0.1
ON	I0.2	ORNOT	I0.2
ON	I0.3	ORNOT	I0.3
=	Q0.0	OUT	Q0.0

5.3.4 : NOR GATE If any contact is open then the output Q0.1 is deenergized.

e.g.:

LADDER PROGRAM.



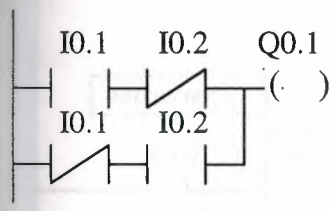
STATEMENT LIST

(Simatic S7)		(Texas Inst.)	
LDN	I0.0	STRNOT	I0.0
AN	I0.1	ANDNOT	I0.1
AN	I0.2	ANDNOT	I0.2
AN	I0.3	ANDNOT	I0.3
=	Q0.0	OUT	Q0.0

5.3.5 : XOR GATE (EXCULUSİVE OR)

e.g.:

LADDER PROGRAM.



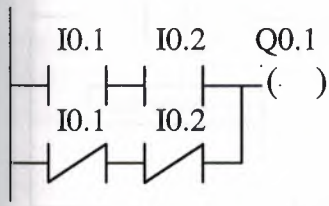
STATEMENT LIST

(Simatic S7)		(Texas Inst.)	
LD	I0.1	STR	I0.1
AN	I0.2	ANDNOT	I0.2
LDN	I0.1	STRNOT	I0.1
A	I0.2	AND	I0.2
OLD		ORSTR	
=	Q0.1	OUT	Q0.1

5.3.6 : XNOR GATE (EXCULUSİVE NOR)

e.g.:

LADDER PROGRAM.



STATEMENT LIST

(Simatic S7)		(Texas Inst.)	
LD	I0.1	STR	I0.1
A	I0.2	AND	I0.2
LDN	I0.1	STRNOT	I0.1
AN	I0.2	ANDNOT	I0.2
OLD		ORSTR	
=	Q0.1	OUT	Q0.1

5.3.7 : TIMER (SIMATIC S7-200)

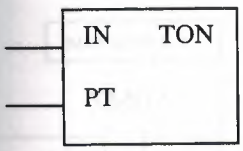
Simatic S7-200 timers are controlled with a singel enabling input and have a current value that maintains the elapsed time since the timer was enabled.

The timers also have a present time value *(PT)* that is compared to the current value each time the current value is updated and a timer bit is set/reset based upon the result of the comparision of current value to the present time value. When the current value is greater then or equal to the present time value the timer bit *(T)* is turned on. Otherwice the T bit is turned off. Timing stop when the corrent value reachas a max value.

When a timer is reset, it is current value is set to zero and it is T bit turned off. Timers can be reset with using the *RESET* instruction (This is the only way to reset a Timer on Retentive Delay *TONR* timer).

- **TİMER ON DELAY (TON)**

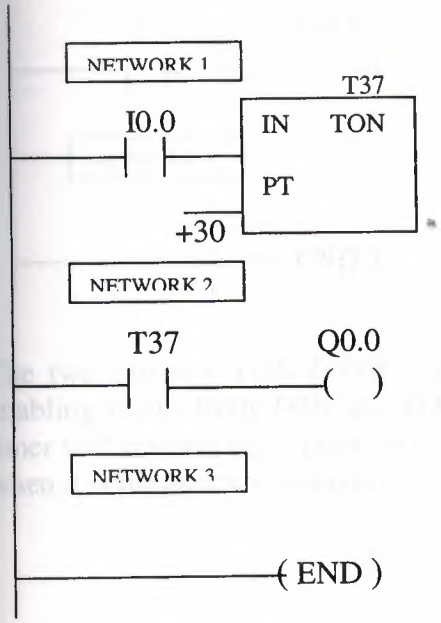
The on delay timer *(TON)* box times up the maximum value when the enabling input comes on (activated). When the current value of the timer > the present time *(PT)* the timer bit turns on. Itresets when the enabling input goes off. Timing stops upon reaching the maximum value.



	CPU 212/214	CPU 214
1 ms	T32	T96
10 ms	T33-T36	T97-T100
100 ms	T37-T63	T101-T127

e.g.:

LADDER PROGRAM.



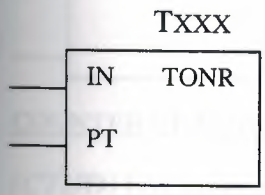
STATEMENT LIST
(Simatic S7)

```
LD    I0.0
TON   T37,+30
LD    T37
=     Q0.0
MEND
```

I0.0 activates T37 after 30*100 ms = 3 sec. T37 will be on and Q0.0 will be activated.

TIMER RETENTIVE ON DELAY

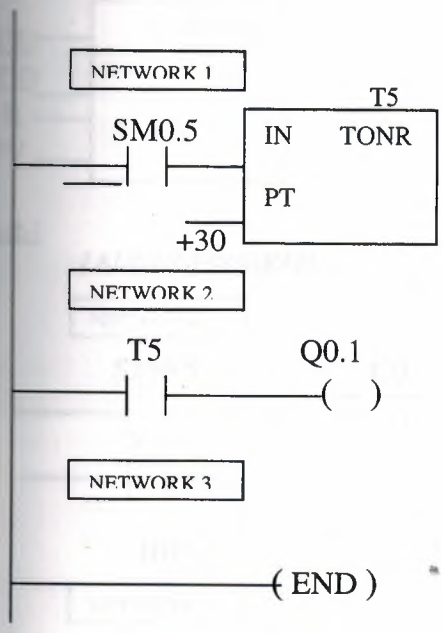
Description of operating on delay timer, times up to the max value when the enabling input is activated when the current value of the timer is greater than of equal to the present time value the timer bit turns on.Timing stops when the enabling input goes off or upon reaching maximum value.



	CPU 212/214	CPU 214
1 ms	T0	T64
10 ms	T1-T4	T65-T68
100 ms	T5-T31	T69-T95

e.g.:

LADDER PROGRAM.



STATEMENT LIST
(Simatic S7)

```
LD      SM0.5
TONR    T5,+30
LD      T5
=       Q0.1
MEND
```

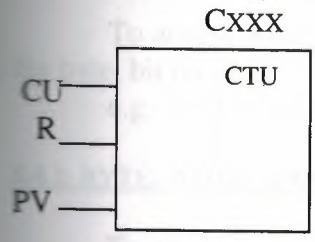
Due to the delay caused by SM0.5 T5 bit will be activated after 6 sec And Q0.1 will be activeted.

The two timers (*TON-TONR*) differ in the ways that they react to the state of the enabling input. Both *TON* and *TONR* time up while the enabling input is off. A *TON* timer will automatically reset and *TONR* timer will not reset. It is converient to use *TONR* when it is necessary to accumulate a number of timed intervals.

5.3.8 : COUNTER

COUNTER UP COUNTER

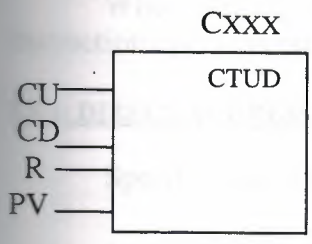
The count up *(CTU)* box counts up to maximum value on the rising edge of the count up input.When the current value (*xxx*) is > to the present value *(PV)* the counter bit (*Cxxx*) turn on.It stops counting upon reaching value (32,767)



CPU 212	CPU 214
C0-63	C0-127

COUNTER UP/DOWN COUNTER

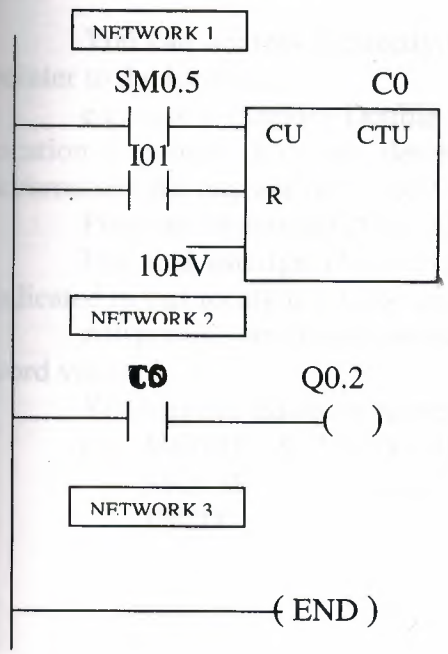
(CTUD) box counters up on rising edge of the count up *(CU)* input or it count down on the rising edge of the count down *(CD)* input when the current value of *(CXXX)* is > the present value *(PV)* the counter bit turns on.



CPU 212	CPU 214
C0-63	C0-127

e.g.:

LADDER PROGRAM.



STATEMENT LIST
(Simatic S7)

```
LD      SM0.5
LD      I0.1
CTU     C0,+10
LD      C0
=       Q0.2
MEND
```

At the beginning order to reset the counter I0.1 input should be activated SM0.5 clock sends pulses and above the CO normally open control is activated as the PV(10) is reached and CO normally open contact activates the output Q0.2 and the program ends.

5.4:ACCESSING DATA MEMORY

To use a memory location, address that location using memory type and number memory areas can be accessed either, as a bit, byte, word, double word.

5.4.1: BIT ACCESS

To access a bit, specify the address of the bit which consits of area identified and the byte, bit number. Zero is the first address for all data areas.

e.g.: I0.0 bit address is a decimal number from 0-7

5.4.2: BYTE, WORD or DOUBLE WORD ACCESS

To access a byte, word or double word specify the address which consist of and area identifier, a letter signifiying data size and the address number.

e.g.: VB200 access V memory location.

5.5: ADDRESSING MODES

When writing the program you can use either of two modes of addressing instruction operands direct or indirect.

5.5.1: DIRECT ADDRESSING

Specifies the memory area and the address

e.g.: VW 790 refers to location 790 in V memory.

5.5.2: INDIRECT ADDRESS

You can address indirectly the data types I,Q,M,T,C and V to the this create a pointer to the location.

e.g.: use a memory Double word (MOVD) insruction to move the address of a location (pointer) to the desired destination. Used only V memory location or accumulator for reqister AC1, AC2 & AC3 as the detiration address.

Place an ampersand (*) at the beginning of the pointer address.

Use and asterigk (*) before the destiration address to indicate the address to indicated in this location is to be used instead of two value.

All pointer are double word values. We them to access byte, word and double word values.

You can not indirectly accress bit values.

```
e.g.: MOVD.  &  VB200, ACI
      MOVW   *  ACI, ACO
      INCD      ACI
```

SIMATIC S7-200
Quick Reference Card

Interrupts

Priority Group	Event	Description	Priority in Group
Communication Interrupts (Highest Priority)	8	Port 0: Receive character [1][2][3][4]	0
	9	Port 0: Transmit [1][2][3][4]	0
	23	Port 0: Receive message [3][4]	0
	24	Port 1: Receive message [4]	1
	25	Port 1: Receive character [4]	1
	26	Port 1: Transmit complete [4]	1
I/O Interrupts (Middle Priority)	0	Rising edge, I0.0* [1][2][3][4]	0
	2	Rising edge, I0.1 [2][3][4]	1
	4	Rising edge, I0.2 [2][3][4]	2
	6	Rising edge, I0.3 [2][3][4]	3
	1	Falling edge, I0.0* [1][2][3][4]	4
	3	Falling edge, I0.1 [2][3][4]	5
	5	Falling edge, I0.2 [2][3][4]	6
	7	Falling edge, I0.3 [2][3][4]	7
	12	HSC0 = preset value* [1][2][3][4]	0
	13	HSC1 = preset value [2][3][4]	8
	14	HSC1 direction change [2][3][4]	9
	15	HSC1 external reset [2][3][4]	10
	16	HSC2 = preset value [2][3][4]	11
	17	HSC2 direction change [2][3][4]	12
	18	HSC2 external reset [2][3][4]	13
	19	PLS0 [2][3][4]	14
	20	PLS1 [2][3][4]	15
Timed Interrupts (Lowest Priority)	10	Timed 0 [1][2][3][4]	0
	11	Timed 1 [2][3][4]	1
	21	T32 = preset [3][4]	2
	22	T96 = preset [3][4]	3

* If event 12 is attached to an interrupt, then event 0 and event 1 cannot be attached to interrupts.

[1] CPU 212 [2] CPU 214 [3] CPU 215 [4] CPU 216

Special Memory Bits

SM0.0	Always On	SM1.0	Result of operation = 0
SM0.1	First Scan	SM1.1	Overflow or illegal value
SM0.2	Retentive data loss	SM1.2	Negative result
SM0.3	Power up	SM1.3	Division by 0
SM0.4	30 s off / 30 s on	SM1.4	Table full
SM0.5	0.5 s off / 0.5 s on	SM1.5	Table empty
SM0.6	Off 1 scan / on 1 scan	SM1.6	BCD to binary conversion error
SM0.7	Switch in RUN position	SM1.7	ASCII to hex conversion error

High-Speed Counter Modes

Counter		Inputs			
HSC0	Maximum 2 kHz [1][2][3][4]	I0.0			
HSC1	7 kHz [2] 20 kHz [3][4]	I0.6	I0.7	I1.0	I1.1
HSC2	7 kHz [2] 20 kHz [3][4]	I1.2	I1.3	I1.4	I1.5

Mode	Description	Clock		Reset	Start
0 to 2	Single phase with internal direction	Up/Down: 0, 1, 2		1, 2	2
3 to 5	Single phase with external direction	Up/Down: 3, 4, 5	Direction: 3, 4, 5	4, 5	5
6 to 8	Two phase	Up: 6, 7, 8	Down: 6, 7, 8	7, 8	8
9 to 11	Quadrature A/B	A: 9, 10, 11	B: 9, 10, 11	10, 11	11

[1] CPU 212 [2] CPU214 [3] CPU 215 [4] CPU 216

Description	Range Limit				Accessible as...			
	212	214	215	216	Bit	Byte	Word	DWord
Total Program Size	512 W	2048 W	4096 W	4096 W				
Total Data Size	512 W	2048 W	2560 W	2560 W				
Variable memory	0-1023	0-4095	0-5119	0-5119	Vx.y	VBx	VWx	VDx
Input Image Register	0-7	0-7	0-7	0-7	Ix.y	IBx	IWx	IDx
Output Image Register	0-7	0-7	0-7	0-7	Qx.y	QBx	QWx	QDx
Analog Inputs	0-30	0-30	0-30	0-30			AIWx	
Analog Outputs	0-30	0-30	0-30	0-30			AQWx	
M memory	0-15	0-31	0-31	0-31	Mx.y	MBx	MWx	MDx
Special memory	0-45	0-85	0-194	0-194	SMx.y	SMBx	SMWx	SMDx
Retentive Timers 1 ms	0	0,64	0,64	0,64	Tx		Tx	
Retentive Timers 10 ms	1-4	1-4, 65-68	1-4, 65-68	1-4, 65-68	Tx		Tx	
Retentive Timers 100 ms	5-31	5-31, 69-95	5-31, 69-95	5-31, 69-95	Tx		Tx	
On/Delay Timers 1 ms	32	32, 96	32, 96	32, 96	Tx		Tx	
On/Delay Timers 10 ms	33-36	33-36, 97-100	33-36, 97-100	33-36, 97-100	Tx		Tx	
On/Delay Timers 100 ms	37-63	37-63, 101-127	37-63, 101-255	37-63, 101-255	Tx		Tx	
Counters	0-63	0-127	0-255	0-255	Cx		Cx	
High Speed Counter	0	0-2	0-2	0-2				HCx
Accumulators	0-3	0-3	0-3	0-3		ACx	ACx	ACx
Sequence Control Relay (SCR)	0-7	0-15	0-31	0-31	Sx.y	SBx	SWx	SDx
Jump/Labels	0-63	0-255	0-255	0-255				
Call/Subroutine	0-15	0-63	0-63	0-63				
Interrupt Routines	0-31	0-127	0-127	0-127				
Interrupt Events	0,1,8-10, 12	0-20	0-23	0-26				
PID Loops	N/A	N/A	0-7	0-7				
Ports	Port 0	Port 0	Port 0 , DP Port	Port 0, Port 1				

Boolean Instructions

		Load
		Load Immediate
		Load Not
		Load Not Immediate
		AND
		AND Immediate
		AND Not
		AND Not Immediate
		OR
		OR Immediate
		OR Not
		OR Not Immediate
	N1, N2	Load result of Byte Compare N1 (=, >=, or <=) N2
	N1, N2	AND result of Byte Compare N1 (=, >=, or <=) N2
	N1, N2	OR result of Byte Compare N1 (=, >=, or <=) N2
	N1, N2	Load result of Word Compare N1 (=, >=, or <=) N2
	N1, N2	AND result of Word Compare N1 (=, >=, or <=) N2
	N1, N2	OR result of Word Compare N1 (=, >=, or <=) N2
	N1, N2	Load result of DWord Compare N1 (=, >=, or <=) N2
	N1, N2	AND result of DWord Compare N1 (=, >=, or <=) N2
	N1, N2	OR result of DWord Compare N1 (=, >=, or <=) N2
	N1, N2 [1]	Load result of Real Compare N1 (=, >=, or <=) N2
	N1, N2 [1]	AND result of Real Compare N1 (=, >=, or <=) N2
	N1, N2 [1]	OR result of Real Compare N1 (=, >=, or <=) N2
		Stack Negation
		Detection of Rising Edge
		Detection of Falling Edge
	N	Assign Value
	N	Assign Value Immediate
	S_BIT, N	Set bit Range
	S_BIT, N	Reset bit Range
	S_BIT, N	Set bit Range Immediate
	S_BIT, N	Reset bit Range Immediate

Math, Increment, and Decrement Instructions

	IN1, OUT	Add Integer, DWord or Real IN1+OUT=OUT
	IN1, OUT	Subtract Integer, DWord, or Real OUT–IN1=OUT
	IN1, OUT	Multiply Integer or Real IN1 * OUT = OUT
	IN1, OUT	Divide Integer or Real OUT / IN1 = OUT
SQRT	IN, OUT [1]	Square Root
INCB	OUT [2]	Increment Byte, Word or DWord
INCW	OUT	
INCD	OUT	
DECB	OUT [2]	Decrement Byte, Word, or DWord
DECW	OUT	
DECD	OUT	
PID	Table, Loop [2]	PID Loop

Timer and Counter Instructions

TON	Txxx, PT	On Delay Timer
TONR	Txxx, PT	Retentive On Delay Timer
CTU	Cxxx, PV	Count Up
CTUD	Cxxx, PV	Count Up/Down

Real Time Clock Instructions

TODR	T [1]	Read Time of Day clock
TODW	T [1]	Write Time of Day clock

Program Control Instructions

END		Conditional End of Program
MEND		Main Program End of Program
STOP		Transition to STOP Mode
WDR		WatchDog Reset (300 ms)
JMP	N	Jump to defined Label
LBL	N	Define a Label to Jump to
CALL	N	Call a Subroutine
SBR	N	Define a Subroutine to be Called
CRET		Conditional Return from SBR
RET		Unconditional Return from SBR
FOR	Index, Initial, Final [1]	For/Next Loop
NEXT	[1]	
LSCR	N	Load, Transition, and End Sequence Control Relay Segment
SCRT	N	
SCRE		

Move, Shift, Rotate, and Fill Instructions

MOVB	IN, OUT	Move Byte, Word, DWord, Real
MOVW	IN, OUT	
MOVD	IN, OUT	
MOVR	IN, OUT [1]	
BMB	IN, OUT, N	Block Move Byte, Word, DWord
BMW	IN, OUT, N	
BMD	IN, OUT, N [2]	
SWAP	IN	Swap Bytes
SHRB	Data, S_bit, N	Shift Register Bit
SRB	OUT, N [2]	Shift Right Byte, Word, DWord
SRW	OUT, N	
SRD	OUT, N	
SLB	OUT, N [2]	Shift Left Byte, Word, DWord
SLW	OUT, N	
SLD	OUT, N	
RRB	OUT, N [2]	Rotate Right Byte, Word, DWord
RRW	OUT, N	
RRD	OUT, N	
RLB	OUT, N [2]	Rotate Left Byte, Word, DWord
RLW	OUT, N	
RLD	OUT, N	
FILL	IN, OUT, N	Fill memory space with pattern

Logic Operations

ALD		And for combinations
OLD		Or for combinations
LPS		Logic Push (stack control)
LRD		Logic Read (stack control)
LPP		Logic Pop (stack control)
ANDB	IN1, OUT [2]	Logical And of Byte, Word, and DWord
ANDW	IN1, OUT	
ANDD	IN1, OUT	
ORB	IN1, OUT [2]	Logical Or of Byte, Word, and DWord
ORW	IN1, OUT	
ORD	IN1, OUT	
XORB	IN1, OUT [2]	Logical XOr of Byte, Word, and DWord
XORW	IN1, OUT	
XORD	IN1, OUT	
INVB	OUT [2]	Invert Byte, Word and DWord (1's complement)
INVW	OUT	
INVD	OUT	

Table, Find, and Conversion Instructions

ATT	Data, Table [1]	Add data to table
LIFO	Table, Data [1]	Get data from table
FIFO	Table, Data [1]	
FND=Scr, Patrn, Indx	[1]	Find data value in table that matches comparison
FND<>Scr, Patrn, Indx	[1]	
FND< Scr, Patrn, Indx	[1]	
FND> Scr, Patrn, Indx	[1]	
BCDI	OUT	Convert BCD to Integer
IBCD	OUT	Convert Integer to BCD
DTR	IN, OUT [1]	Convert DWord to Real
TRUNC	IN, OUT [1]	Convert Real to DWord
ATH	IN, OUT, LEN	Convert ASCII to HEX
HTA	IN, OUT, LEN	Convert HEX to ASCII
DECO	IN, OUT	Decode
ENCO	IN, OUT	Encode
SEG	IN, OUT	Generate 7–segment pattern

Interrupt

INT	N	Beginning of Interrupt routine
CRETI		Conditional Return form Interrupt
RETI		Return from Interrupt
ENI		Enable Interrupts
DISI		Disable Interrupts
ATCH	INT, EVENT	Attach Interrupt routine to event
DTCH	EVENT	Detach event

Communication

XMT	TABLE, PORT	Freeport transmission
RCV	TABLE, PORT [2]	Freeport receive message
NETR	TABLE,PORT [1]	Network Read
NETW	TABLE,PORT [1]	Network Write

High Speed Instructions

HDEF	HSC, Mode	Define High Speed Counter mode
HSC	N	Activate High Speed Counter
PLS	X [1]	Pulse Output

Instructions are valid for the individual S7-200 PLCs as marked according to the following key:

[1] 214, 215, and 216 only

[2] 215 and 216 only

If not marked, the instructions are valid for all S7-200 PLCs.

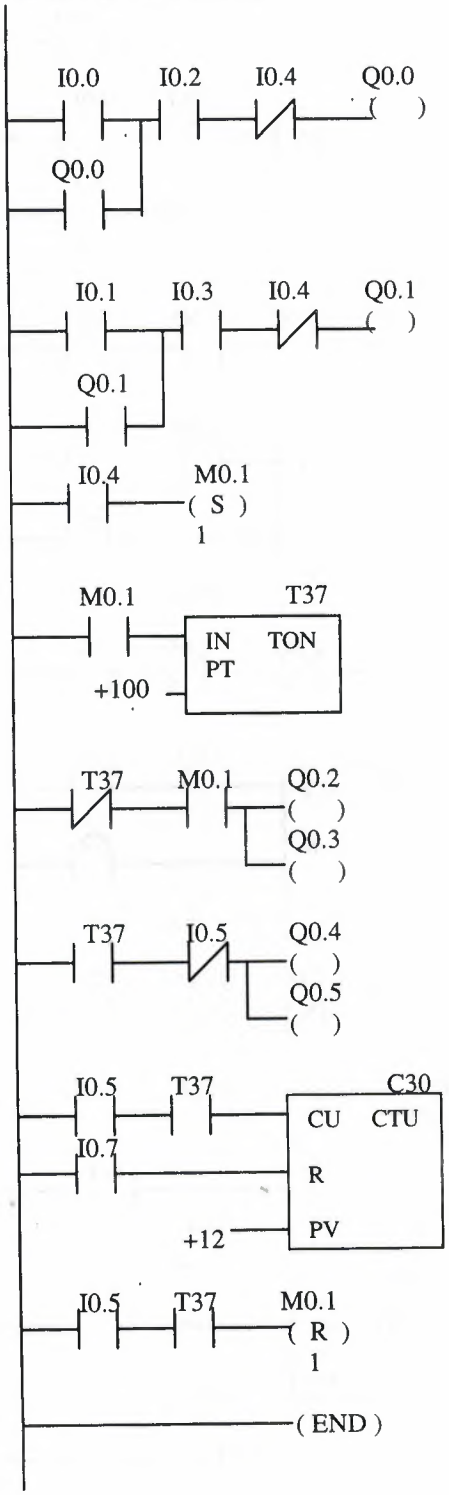
5.6 : Sample Programs

SAMPLE PROGRAM 1: *(POINT MIXER)*

STATEMENT LIST (SIMATIC S7)

```
NETWORK 1
LD      I0.0
O       Q0.0
A       I0.2
AN      I0.4
=       Q0.0

NETWORK 2
LD      I0.1
O       Q0.1
A       I0.3
AN      I0.4
=       Q0.1

NETWORK 3
LD      I0.4
S       M0.1, 1

NETWORK 4
LD      M0.1
TON     T37, +100

NETWORK 5
LDN     T37
A       M0.1
=       Q0.2
=       Q0.3

NETWORK 6
LD      T37
AN      I0.5
=       Q0.4
=       Q0.5

NETWORK 7
LD      I0.5
A       T37
LD      I0.7
CTU     C30, +12

NETWORK 8
LD      I0.5
A       T37
R       M0.1, 1

NETWORK 9
MEND
```

LADDER PROGRAM

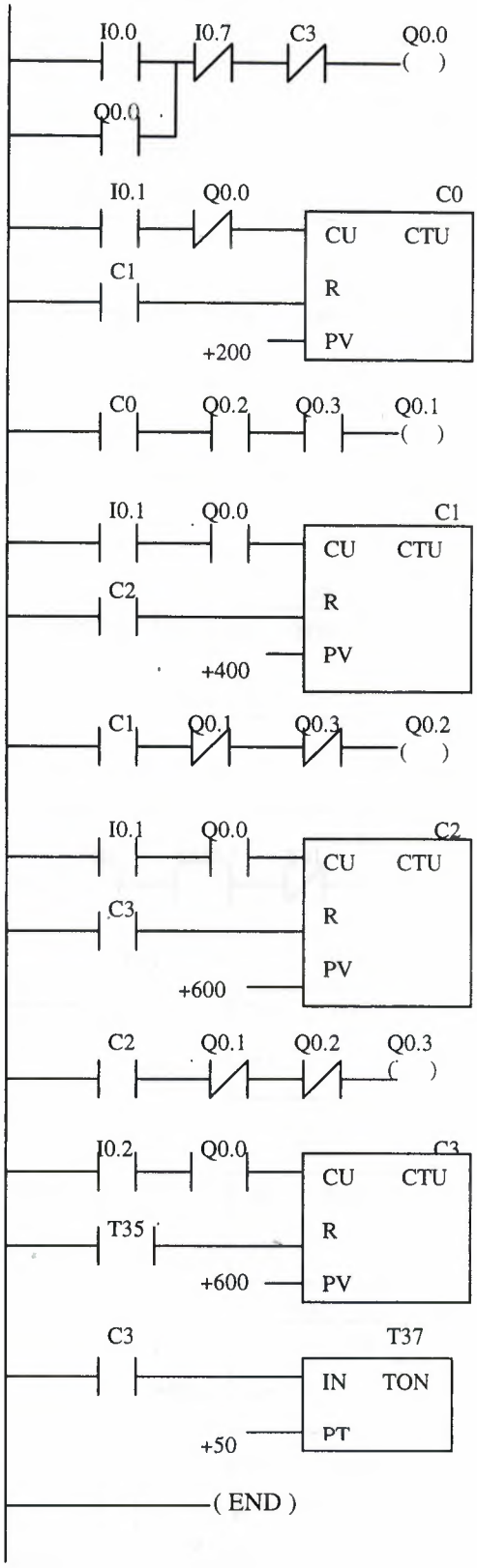


SAMPLE PROGRAM 2:

STATEMENT LIST (SIMATIC S7)

```
NETWORK 1
LD      I0.0
O       Q0.0
AN      I0.7
AN      C3
=       Q0.0

NETWORK 2
LD      I0.1
AN      Q0.0
LD      C1
CTU     C0, 200

NETWORK 3
LD      C0
A       Q0.2
A       Q0.3
=       Q0.1

NETWORK 4
LD      I0.1
A       Q0.0
LD      C2
CTU     C1, 400

NETWORK 5
LD      C1
AN      Q0.1
AN      Q0.3
=       Q0.2

NETWORK 6
LD      I0.1
A       Q0.0
LD      C3
CTU     C2,600

NETWORK 7
LD      C2
AN      Q0.1
AN      Q0.2
=       Q0.3

NETWORK 8
LD      I0.2
A       Q0.0
LD      T35
CTU     C3, 600

NETWORK 9
LD      C3
TON     T35, +50

NETWORK 10
MEND
```

LADDER PROGRAM

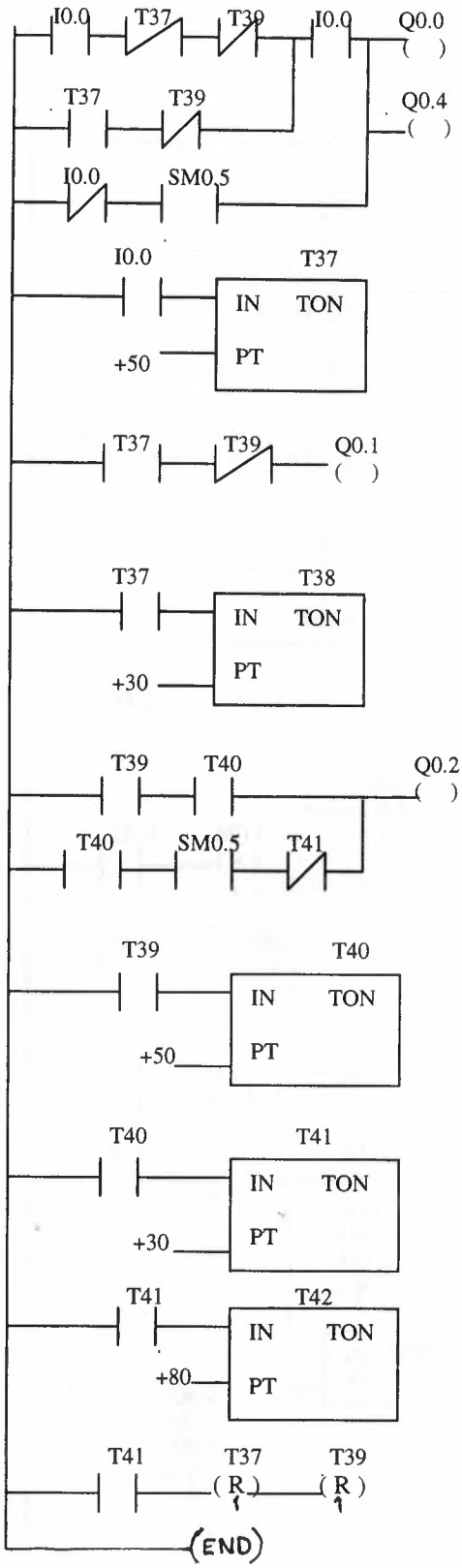


SAMPLE PROGRAM 3:

STATEMENT LIST (SIMATIC S7)

```
NETWORK 1
LD      I0.0
AN      T37
AN      T39
LD      T37
AN      T39
OLD
A       I0.0
LDN     I0.0
A       SM0.5

OLD
=       Q0.0
=       Q0.4

NETWORK 2
LD      I0.0
TON     T37, +50

NETWORK 3
LD      T37
AN      T39
=       Q0.1
NETWORK 4
LD      T37
TON     T38, +30

NETWORK 5
LD      T39
AN      T40
LD      T40
A       SM0.5
AN      T41
OLD
=       Q0.2

NETWORK 6
LD      T39
TON     T40, +50

NETWORK 7
LD      T40
TON     T41, +30

NETWORK 8
LD      T41
TON     T42, +80

NETWORK 9
LD      T41
R       T37, 1
R       T39, 1

NETWORK 10
MEND
```

LADDER PROGRAM

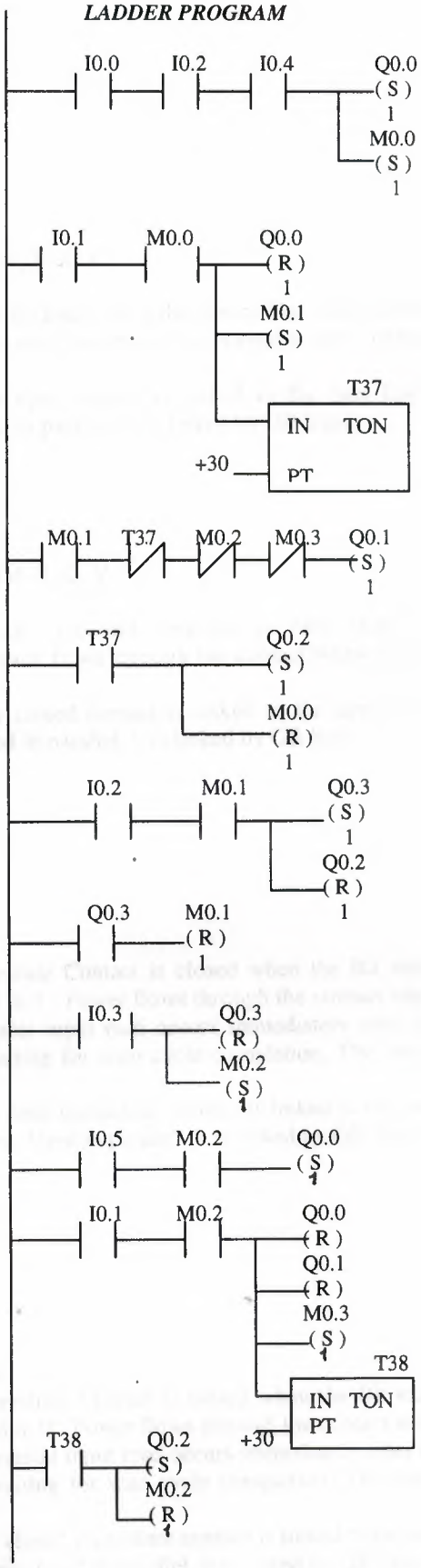


SAMPLE PROGRAM 4:

STATEMENT LIST (SIMATIC S7)

```
NETWORK 1
LD      I0.0
A       I0.2
A       I0.4
S       Q0.0 ,1
S       M0.0, 1

NETWORK 2
LD      I0.1
A       M0.0
R       Q0.0
S       M0.1, 1
TON     T37, +30

NETWORK 3
LD      M0.1
AN      T37
AN      M0.2
AN      M0.3
S       Q0.1

NETWORK 4
LD      T37
S       Q0.2, 1
R       M0.0, 1

NETWORK 5

LD      I0.2
A       M0.1
S       Q0.3, 1
R       Q0.2, 1

NETWORK 6
LD      Q0.3
R       M0.1, 1

NETWORK 7
LD      I0.3
R       Q0.3, 1
S       M0.2, 1

NETWORK 8
LD      I0.5
A       M0.2
S       Q0.0, 1

NETWORK 9
LD      I0.1
A       M0.2
R       Q0.0, 1
R       Q0.1, 1
S       M0.3, 1
TON     T38, +30

NETWORK 10
LD      T38
S       Q0.2
R       M0.2
```

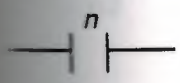
LADDER PROGRAM



6.1 : *Ladder Instruction Set:*

Normally Open Contact

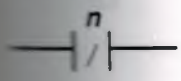
Symbol:



Operands:
n (bit): I, Q, M, SM, T, C, V
Description of operation:
The Normally Open Contact is closed when the scanned bit value stored at address n is equal to 1 . Power flows through a normally open contact when closed (activated).
Used in series, a normally open contact is linked to the next LAD element by AND logic. Used in parallel, it is linked by OR logic.

Normally Closed Contact

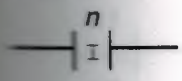
Symbol:



Operands:
n (bit): I, Q, M, SM, T, C, V
Description of operation:
The Normally Closed Contact is closed when the bit value stored at address n is equal to 0 . Power flows through the contact when closed (deactivated).
Used in series, a normally closed contact is linked to the next LAD element by AND logic. Used in parallel, it is linked by OR logic.

Normally Open Immediate Contact

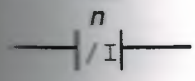
Symbol:



Operands:
n (bit)
Description of operation:
The Normally Open Immediate Contact is closed when the Bit value stored at address n is equal to 1 . Power flows through the contact when closed (activated). A physical input read occurs immediately after the coil is scanned without waiting for scan cycle completion. The image register is not updated.
Used in series, a normally open immediate contact is linked to the next LAD element by AND logic. Used in parallel, it is linked by OR logic

Normally Closed Immediate Contact

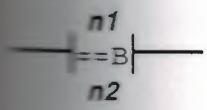
Symbol:



Operands:
n (bit): I
Description of operation:
The Normally Closed Immediate Contact is closed when the Bit value stored at address n is equal to 0 . Power flows through the contact when closed (deactivated). A physical input read occurs immediately after the coil is scanned without waiting for scan cycle completion. The image register is not updated.
Used in series, a normally closed immediate contact is linked to the next LAD element by AND logic. Used in parallel, it is linked by OR logic.

Compare Byte Equal Contact

Symbol:



Operands:

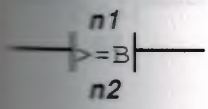
n1, n2 (unsigned byte): VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The Compare Byte Equal Contact is closed when the byte value stored at address n1 is equal to the byte value stored at address n2 . Power flows through the contact when closed.

Compare Byte Greater Than Or Equal Contact

Symbol:



Operands:

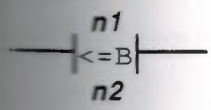
n1, n2 (unsigned byte): VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The Compare Byte Greater Than or Equal Contact is closed when the byte value stored at address n1 is greater than or equal to the byte value stored at address n2 . Power flows through the contact when closed.

Compare Byte Less Than Or Equal Contact

Symbol:



Operands:

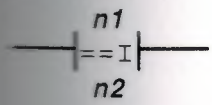
n1, n2 (unsigned byte): VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The Compare Byte Less Than or Equal Contact is closed when the byte value stored at address n1 is less than or equal to the byte value stored at address n2 . Power flows through the contact when closed

Compare Integer Equal Contact

Symbol:



Operands:

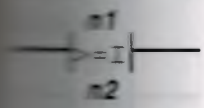
n1, n2 (signed integer word): VW, T,C,IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:

The Compare Integer Equal Contact is closed when the signed integer word value stored at address n1 is equal to the signed integer word value stored at address n2 . Power flows through the contact when closed.

Compare Integer Greater Than Or Equal Contact

Symbol:



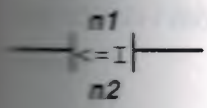
Operands:

n1, n2 (signed integer word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:
The Compare Integer Greater Than or Equal Contact is closed when the signed integer word value stored at address n1 is greater than or equal to the signed integer word value stored at address n2 . Power flows through the contact when closed

Compare Integer Less Than Or Equal Contact

Symbol:



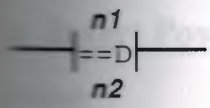
Operands:

N1, n2 (signed integer word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:
The Compare Integer Less Than or Equal Contact is closed when the signed integer word value stored at address n1 is less than or equal to the signed integer word value stored at address n2 . Power flows throughthecontactwhenclosed.

Compare Double Integer Equal Contact

Symbol:



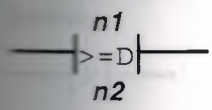
Operands:

n1, n2 (signed integer double word):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC

Description of operation:
The Compare Double Integer Equal Contact is closed when the double word value stored at address n1 is equal to the double word value stored at address n2 . Power flows through the contact when closed

Compare Double Integer Greater Than Or Equal Contact

Symbol:



Operands:

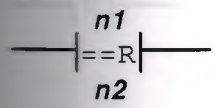
n1, n2 (signed integer double word):	VD, ID, QD, MD, SMD,AC HC, Constant, *VD, *AC

Description of operation:
Compare Double Integer Greater Than Or Equal Contact is closed when the double word value stored at address n1 is greater than or equal to the double word value stored at address n2 . Power flows through the contact when closed.

Compare Real Equal Contact

Note: CPU 214 only.

Symbol:



Operands:

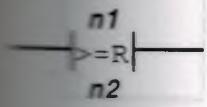
n1, n2 (real):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC

Description of operation:
The Compare Real Equal Contact is closed when the real value stored at address n1 is equal to the real value stored at address n2 . Power flows through the contact when closed.

Compare Real Greater Than Or Equal Contact

Note: CPU 214 only.

Symbol:



Operands:

n1, n2 (Dword): VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC

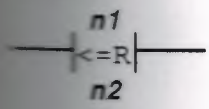
Description of operation:

Compare Real Greater Than Or Equal Contact is closed when the real value stored at address n1 is greater than or equal to the real value stored at address n2 . Power flows through the contact when closed.

Compare Real Less Than Or Equal Contact

Note: CPU 214 only.

Symbol:



Operands:

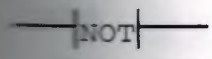
n1, n2 (Dword): VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC

Description of operation:

The Compare Real Less Than Or Equal Contact is closed when the real value stored at address n1 is less than or equal to the real value stored at address n2 . Power flows through the contact when closed.

Invert Power Flow Contact

Symbol:



Operands:

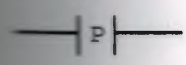
(none)

Description of operation:

The NOT (Invert Power Flow) contact changes the state of power flow. If power flow reaches the Not contact, then it stops. When power flow does not reach the Not contact, it sources power flow

Positive Transition Contact

Symbol:



Operands:

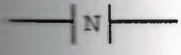
(none)

Description of operation:

The Positive Transition Contact allows power to flow for one scan, for each off-to-on transition .

Negative Transition Contact

Symbol:



Operands:

(none)

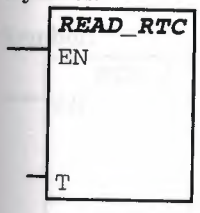
Description of operation:

The Negative Transition Contact allows power to flow for one scan, for each on-to-off transition .

Read Real Time Clock

Note: Real Time Clock instructions are supported by the CPU 214 only.

Symbol:



Operands:

T (byte): VB, IB, QB, MB, SMB, *VD, *AC

Description of operation:

The Read Real Time Clock (READ_RTC) box reads the current time and date from the clock and loads it in an 8-byte buffer (T).

Example Memory Data Starting at VB400:

READ_RTC (Clock is read)

VB400	95	Year
VB401	03	Month
VB402	24	Day
VB403	08	Hour
VB404	00	Minute
VB405	00	Second
VB406	00	
VB407	06	Day of Week

24-Mar-95
8:00:00
Friday

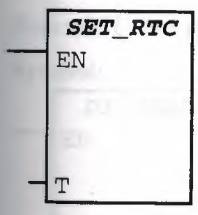
Note: The time of day clock initializes the following date and time after extended power outages or memory has been lost:

Date:	01-Jan-90
Time:	00:00:00
Day of Week	Sunday

Set Real Time Clock

Note: Real Time Clock instructions are supported by the CPU 214 only.

Symbol:



Operands:

T (byte): VB, IB, QB, MB, SMB, *VD, *AC

Description of operation:

The Set Real Time Clock (SET_RTC) box writes the current time and date loaded in an 8-byte buffer (T) to the clock.

Example Memory Data Starting at VB400:

SET_RTC (New value is written to clock)

VB400	96	Year
VB401	03	Month
VB402	24	Day
VB403	08	Hour
VB404	00	Minute
VB405	00	Second
VB406	00	
VB407	06	Day of Week

24-Mar-96
8:00:00
Friday

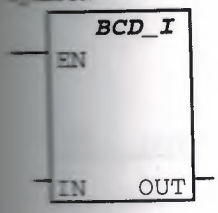
Note: The time of day clock initializes the following date and time after extended power outages or memory has been lost:

Date:	01-Jan-90
Time:	00:00:00
Day of Week	Sunday

Note: Do not use the READ_RTC / SET_RTC instructions in both the main program and in an interrupt routine. If you do this and the clock instruction is executing when the the interrupt that also executes the clock instruction occurs, then the clock instruction in the interrupt routine is not executed. SM4.5 is then set, indicating that two simultaneous accesses to the clock were attempted

BCD to Integer

Symbol:



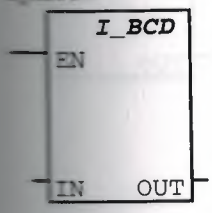
Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:
The Convert BCD to Integer (BCD_I) box converts the BCD value (IN) to an integer value (OUT). If the input value contains an invalid BCD digit, the BCD/BIN memory bit (SM1.6) is set.

Integer to BCD

Symbol:



Operands:

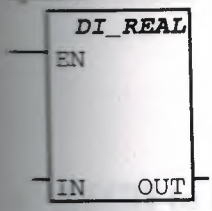
IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:
The Convert Integer to BCD (I_BCD) box converts the integer value (IN) to the BCD value (OUT). If the conversion produces a BCD number greater than 9999, the BCD/BIN memory bit (SM1.6) is set.

Integer Double Word to Real

Note: CPU 214 only.

Symbol:



Operands:

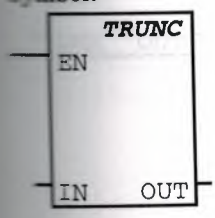
IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:
The Integer Double Word to Real (DI_REAL) instruction converts a 32-bit, signed integer (IN) into a 32-bit real number (OUT).

Truncate

Note: CPU 214 only.

Symbol:

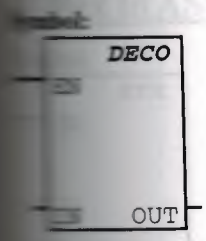


Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:
The Truncate (TRUNC) instruction converts a 32-bit real number (IN) into a 32-bit signed integer (OUT). Only the whole number portion of the real number is converted (round-to-zero).

Decode

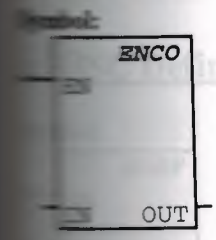


Operands:

IN (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, AQW, *VD, *AC

Description of operation:
The Decode (DECO) box sets the bit in the output word (OUT) that corresponds to the bit number represented by the least-significant nibble (LSN) of the input byte (IN). All other bits of the output word are set to 0.

Encode

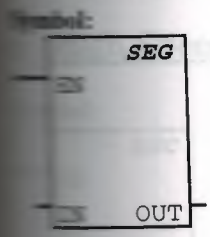


Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, AC, *VD, *AC

Description of operation:
The Encode (ENCO) box writes the bit number (bit #) of the least-significant bit set of the input word (IN) into the least-significant nibble (LSN) of the output byte (OUT).

Segment

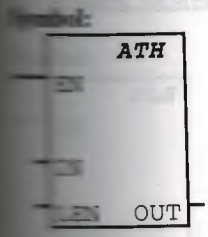


Operands:

IN (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, AC, *VD, *AC

Description of operation:
The Segment (SEG) box generates a bit pattern (OUT) that illuminates the segments of a seven-segment display. The illuminated segments represent the character in the least-significant digit of the input byte (IN).

ASCII to Hex



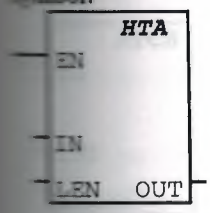
Operands:

LEN (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
IN (byte):	VB, IB, QB, MB, SMB, *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, *VD, *AC

Description of operation:
The ASCII to HEX (ATH) box converts the ASCII string of length LEN, starting with the character IN, to hexadecimal digits starting at the location OUT. The maximum length of the ASCII string is 255 characters.
Legal ASCII characters are the hexadecimal values 30-39, and 41-46. If an illegal ASCII character is encountered, the conversion is terminated, and the NOT_ASCII memory bit (SM1.7) is set.

Hex to ASCII

Symbol:



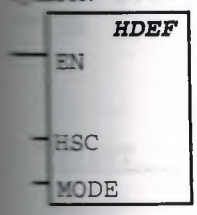
Operands:

LEN (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
IN (byte):	VB, IB, QB, MB, SMB, *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, *VD, *AC

Description of operation:
The HEX to ASCII (HTA) box converts the hexadecimal digits, starting with the input byte IN, to an ASCII string starting at the location OUT. The number of hexadecimal digits to be converted is specified by length LEN. The maximum number of the hexadecimal digits that can be converted is 255.

HSC Definition

Symbol:



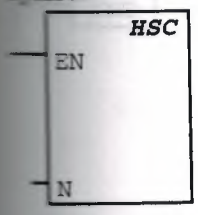
Operands:

HSC (byte):	CPU 212: 0 CPU 214: 0-2
MODE (byte):	CPU 212: 0 CPU 214: 0 (HSC0), 0-11 (HSC1-2)

Description of operation:
When the High-speed Counter Definition (HDEF) box is enabled, the referenced counter (HSC) is assigned a high-speed counter type or MODE. Only one HDEF box may be used per counter.

High Speed Counter

Symbol:



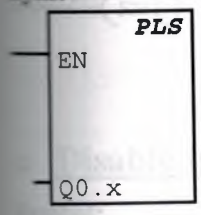
Operands:

N (word):	CPU 212: 0 CPU 214: 0-2

Description of operation:
When the High-speed Counter (HSC) box is enabled, the state of the HSC special memory bits are examined. The HSC operation defined by the special memory bits is then invoked. The parameter N specifies the High-speed Counter number.

Pulse Output

Symbol:



Operands:

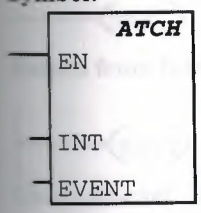
Q0.x (word):	CPU 214: 0-1

Description of operation:
The Pulse Output (PLS) box examines the special memory bits for that pulse output (Q0.x). The pulse operation defined by the special memory bits is then invoked.

Ladder High-speed Operation Instruction Examples

Attach Interrupts

Symbol:



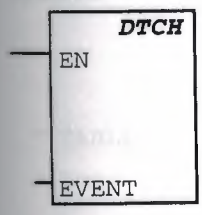
Operands:

INT (byte):	CPU 212: 0-31
	CPU 214: 0-127
EVENT (byte):	CPU 212: 0, 1, 8-10, 12
	CPU 214: 0-20

Description of operation:
The Attach Interrupts (ATCH) box associates an interrupt event (EVENT) with an interrupt routine number (INT), and enables the interrupt event.

Detach Interrupts

Symbol:



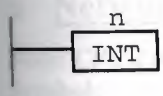
Operands:

EVENT (byte):	CPU 212: 0, 1, 8-10, 12
	CPU 214: 0-20

Description of operation:
The Detach Interrupts (DTCH) box disassociates an interrupt event (EVENT) from all interrupt routines, and disables the interrupt event.

Interrupt Routine

Symbol:



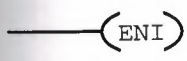
Operands:

n (word):	CPU 212: 0-31
	CPU 214: 0-127

Description of operation:
The Interrupt Routine (INT) label marks the beginning of the interrupt routine (n). The maximum number of interrupts supported by the CPU 212 is 32, and by the CPU 214, 128.

Enable Interrupts

Symbol:



Operands:
(none)
Description:
The Enable Interrupts (ENI) coil globally enables processing of all attached interrupt events.

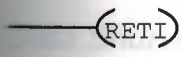
Disable Interrupts

Symbol:

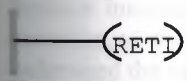
Operands:
(none)
Description:
The Disable Interrupts (DISI) coil globally disables processing of all interruptevents.

Return from Interrupts

Symbol:



Conditional Return from Interrupts

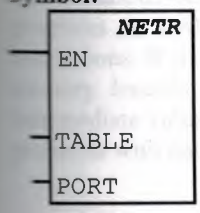


Unconditional Return from Interrupts

Operands:
(none)
Description:
The Conditional Return from Interrupts (RETI) coil returns from an interrupt based upon the condition of the preceding logic.
The Unconditional Return from Interrupts (RETI) coil must be used to terminateeachinterruptroutine.

Network Read

Note: CPU 214 only.
Symbol:

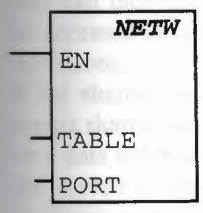


Operands:
TABLE: VB, MB, *VD, *AC
PORT: Constant
(CPU 214: 0)
Description of operation:

The Network Read (NETR) instruction initiates a communication operation to gather data from a remote device through the specified port (PORT), as defined in the description table (TABLE).
You can use the NETR instruction to read up to 16 bytes of information from a remote station, and use the NETW instruction to write up to 16 bytes of information to a remote station. A maximum of eight NETR and NETW instructions may be activated at any one time. For example, you can have four NETR and four NETW instructions, or two NETR andsixNETWinstructions.

Network Write

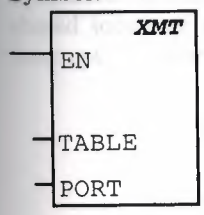
Note: CPU 214 only.
Symbol:



Operands:
TABLE: VB, MB, *VD, *AC
PORT: Constant
(CPU 214: 0)
Description of operation:
The Network Write (NETW) instruction initiates a communication operation to write data to a remote device through the specified port (PORT), as defined in the description table (TABLE).
You can use the NETR instruction to read up to 16 bytes of information from a remote station, and use the NETW instruction to write up to 16 bytes of information to a remote station. A maximum of eight NETR and NETW instructions may be activated at any one time. For example, you can have four NETR and four NETW instructions, or two NETR andsixNETWinstructions.

Transmit

Symbol:



Operands:
TABLE (byte): VB, IB, QB, MB, SMB, *VD, *AC
PORT (byte): 0
Description of operation:
The Transmit (XMT) box invokes the transmission of the data buffer (TABLE). The first entry in the data buffer specifies the number of bytes to be transmitted. PORT specifies the communication port to be used for transmission. It must always be 0.

Data Sharing with Interrupt Events

Because interrupt events are asynchronous to the main user-program, they can occur at any point during execution of the main user-program. When the main program and an interrupt routine share data, you must understand the nature of the problems that can arise and how to avoid such problems.

Data-sharing problems can occur in situation where a sequence of operations are performed in the main program on data stored in a memory location shared by the main program and an interrupt routine. If an intermediate result is stored in the shared memory location, then an interrupt event occurring before the sequence is complete will cause the interrupt routine to be executed with invalid data, or it will corrupt an intermediate value in the main program.

The situations described above apply whether you write your programs in STL or LAD. If you write your programs in LAD, you should also be aware that many LAD instructions produce a sequence of STL instructions. If the LAD instruction is located in the main program and is operating on data stored in a shared memory location, an interrupt event can occur between the execution of the STL instructions, altering intermediate values and making it appear that the LAD instruction executed incorrectly. For techniques to avoid problems with data sharing, see Programming Techniques for Data Sharing .

Programming Techniques for Data Sharing

The following programming techniques should be followed to avoid problems with data sharing between your main program and interrupt routines. These techniques either restrict the way access is made to shared memory locations, or they make instruction sequences using shared memory locations uninterruptible. The appropriate technique depends upon the size of the data being shared (simple elements such as a byte, word, or double-word variable or complex elements such as multiple variables) and the programming language (STL or LAD).

If the shared data is a single byte, word, or double-word variable and your program is written in STL, then make sure that intermediate or temporary values are not stored in shared memory locations. A shared location should be accessed in the main program only as the initial source value or the final destination value in a sequence of operations.
If the shared data is a single byte, word, or double-word variable and your program is written in LAD, then access shared memory locations using a Move instruction. If the main program performs one or more operations on a data value provided by an interrupt routine, the Move instruction must be used to move the data value from the shared memory location to a non-shared memory location or to an accumulator. If the main program performs one or more operations on data in order to provide a value to an interrupt routine, then the last operation must be a Move instruction that moves the data value from an accumulator or non-shared memory location to the shared memory location. Other instructions in the sequence must not directly access the shared memory location.

If the shared data is composed of related bytes, words, or double-words whose values must agree; for example, the pressure and temperature of a gas in a tank, then the interrupt disable/enable instructions, DISI and ENI, must be used to control interrupt routine execution. At the point in your main program (STL or LAD) where operations on shared memory locations are to begin, interrupts must be disabled. Once all actions affecting shared locations are complete, interrupts must be re-enabled. During the time that interrupts are disabled, interrupt routines cannot execute and access shared memory locations.

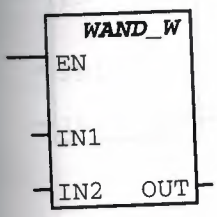
Interrupt Event Priority Table

Interrupt Description (By group priority)	Event #	In-Group Priority	Supported in CPU 212
Comm. (Highest Priority)			
Receive interrupt	8	0	Y
Transmit complete interrupt	9	0*	Y
Discrete (Middle Priority)			
Rising edge, I0.0**	0	0	Y
Rising edge, I0.1	2	1	
Rising edge, I0.2	4	2	
Rising edge, I0.3	6	3	
Falling edge, I0.0**	1	4	Y
Falling edge, I0.1	3	5	
Falling edge, I0.2	5	6	
Falling edge, I0.3	7	7	
HSC0 CV=PV** (*current value = preset value*)	12	0	Y
HSC1 CV=PV (*current value = preset value*)	13	8	
HSC1 direction input changed	14	9	
HSC1 external reset	15	10	
HSC2 CV=PV (*current value = preset value*)	16	11	
HSC2 direction input changed	17	12	
HSC2 external reset	18	13	
PLS0 pulse count complete interrupt	19	14	
PLS1 pulse count complete interrupt	20	15	
Timed (Lowest Priority)			
Timed interrupt 0	10	0	Y
Timed interrupt 1	11	1	

* Since communication is inherently half-duplex, both transmit and receive are the same priority.
**If event 12 (HSC0 CV=PV) is attached to an interrupt, then neither event 0 nor event 1 can be attached to interrupts. Likewise, if either event 0 or 1 is attached to an interrupt, then event 12 cannot be attached to an interrupt.

❑ AND Word

Symbol:



Operands:

IN1, IN2 (word): VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

OUT (word): VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

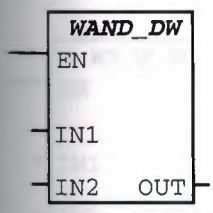
Description of operation:
The AND Word (WAND_W) box ANDs the corresponding bits of the input words IN1 and IN2, and loads the result (OUT) in a word.

Note: When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction isinvalid.

AND Double Word

Symbol:



Operands:

IN1, IN2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

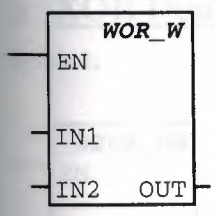
Description of operation:
The AND Double Word (WAND_DW) box ANDs the corresponding bits of the input double words IN1 and IN2, and loads the result (OUT) in a double word.

Note: When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction isinvalid.

OR Word

Symbol:



Operands:

IN1, IN2 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

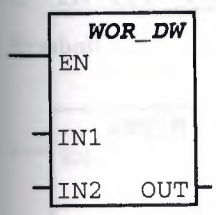
Description of operation:
The OR Word (WOR_W) box ORs the corresponding bits of the input words IN1 and IN2, and loads the result (OUT) in a word.

Note: When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction isinvalid.

OR Double Word

Symbol:



Operands:

IN1, IN2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

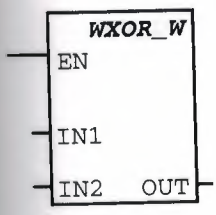
Description of operation:
The OR Double Word (WOR_DW) box ORs the corresponding bits of the input double words IN1 and IN2, and loads the result (OUT) in a double word.

Note: When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction isinvalid.

XOR Word

Symbol:



Operands:

IN1, IN2 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

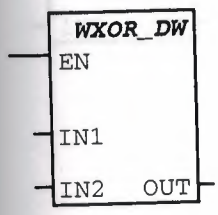
Description of operation:
The Exclusive OR Word (WXOR_W) box XORs the corresponding bits of the input words IN1 and IN2, and loads the result (OUT) in a word.

Note: When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction isinvalid.

XOR Double Word

Symbol:



Operands:

IN1, IN2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

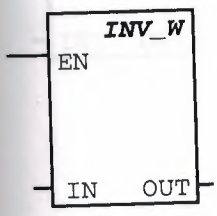
Description of operation:
The Exclusive OR Double Word (WXOR_DW) box XORs the corresponding bits of the input double words IN1 and IN2, and loads the result (OUT) in a double word.

Note: When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction isinvalid.

Invert Word

Symbol:



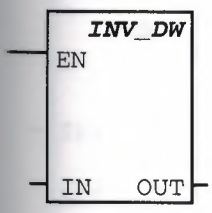
Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:
The Invert Word (INV_W) box takes the ones complement of the input word value (IN) and loads the result in a word value (OUT).

Invert Double Word

Symbol:



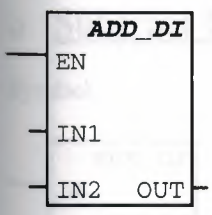
Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:
The Invert Double Word (INV_DW) box takes the ones complement of the input double word value (IN) and loads the result in a double word value (OUT).

Add Double Integer

Symbol:



Operands:

IN1, IN2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:
The Add Double Integer (ADD_DI) box adds two 32-bit integers (IN1, IN2), and produces a 32-bit result (OUT), as is shown in the equation:
IN1 + IN2 = OUT

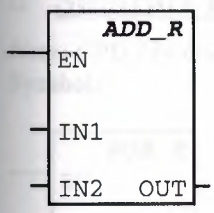
Note: When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction isinvalid.

Add Real

Note: CPU 214 only.

Symbol:



Operands:

IN1, IN2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, SMD, AC, *VD, *AC

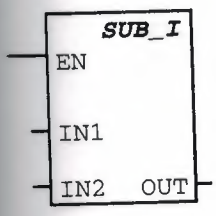
Description of operation:
The Add Real (ADD_R) box adds two 32-bit real numbers (IN1, IN2), and produces a 32-bit real number result (OUT), as is shown in the equation:
IN1 + IN2 = OUT

Note: When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.

Subtract Integer

Symbol:



Operands:

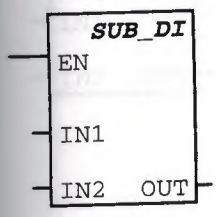
IN1, IN2 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:
The Subtract Integer (SUB_I) box subtracts two 16-bit integers (IN1, IN2), and produces a 16-bit result (OUT), as is shown in the equation:
IN1 – IN2 = OUT
Note: When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction isinvalid.

Subtract Double Integer

Symbol:



Operands:

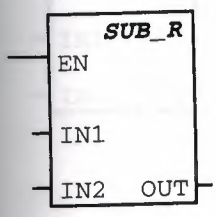
IN1, IN2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:
The Subtract Double Integer (SUB_DI) box subtracts two 32-bit integers (IN1, IN2), and produces a 32-bit result (OUT), as is shown in the equation:
IN1 – IN2 = OUT
Note: When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction isinvalid.

Subtract Real

Note: CPU 214 only.
Symbol:



Operands:

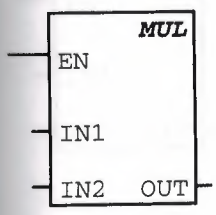
IN1, IN2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, SMD, AC, *VD, *AC

Description of operation:
The Sutract Real (SUB_R) box subtracts two 32-bit real numbers (IN1, IN2), and produces a 32-bit real number result (OUT), as is shown in the equation:
IN1 – IN2 = OUT
Note: When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction isinvalid.

Multiply Integer

Symbol:



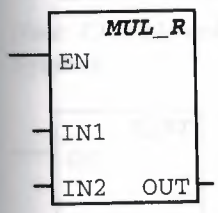
Operands:

IN1, IN2 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:
The Multiply Integer (MUL) box multiplies two 16-bit integers (IN1, IN2), and produces a 32-bit result (OUT), as is shown in the equation:
IN1 * IN2 = OUT
Note: Some overlapping input and output operands are invalid

Multiply Real

Note: CPU 214 only.
Symbol:



Operands:

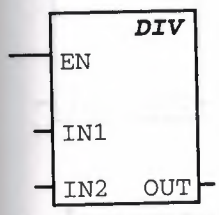
IN1, IN2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, SMD, AC, *VD, *AC

Description of operation:
The Multiply Real (MUL_R) box multiplies two 32-bit real numbers (IN1, IN2), and produces a 32-bit real number result (OUT), as is shown in the equation:
IN1 * IN2 = OUT
Note: When IN1 ≠ OUT and IN2 ≠ OUT:

- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction isinvalid.

Divide Integer

Symbol:



Operands:

IN1, IN2 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

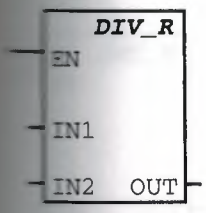
Description of operation:
The Divide Integer (DIV) box divides two 16-bit integers (IN1, IN2), and produces a 32-bit result (OUT) composed of of a 16-bit quotient and a 16-bit remainder, as is shown in the equation:
IN1 / IN2 = OUT
Notes:

- Some overlapping input and output operands are invalid.
- The 32-bit result (OUT) cannot be the same as the second input (IN2).

Divide Real

Note: CPU 214 only.

Symbol:



Operands:

IN1, IN2 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, SMD, AC, *VD, *AC

Description of operation:
The Divide Real (DIV_R) box divides two 32-bit real numbers (IN1, IN2), and produces a 32-bit real number quotient (OUT), as is shown in the equation: .
IN1 / IN2 = OUT

Note: When IN1 ≠ OUT and IN2 ≠ OUT:

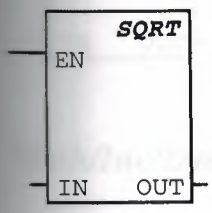
- If IN2 and OUT are direct-addressed operands, and if OUT contains one of the bytes of IN2, then the instruction is invalid.
- If IN2 is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.

Note: IN2 = OUT is not valid for Ladder programming.

Square Root Real

Note: CPU 214 only.

Symbol:



Operands:

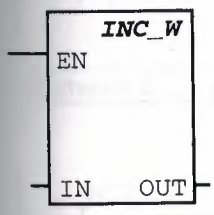
IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:
The Square Root of Real Numbers (SQRT) box takes the square root of a 32-bit real number (IN) and produces a 32-bit real number result (OUT), as is shown in the equation:

$$\sqrt{IN} = OUT$$

Increment Word

Symbol:



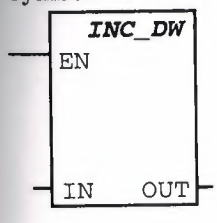
Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:
The Increment Word (INC_W) box adds 1 to the input word value (IN) and loads the result in a word value (OUT), as is shown in the equation:
IN+1=OUT

Increment Double Word

Symbol:



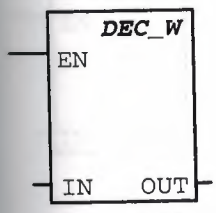
Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:
The Increment Double Word (INC_DW) box adds 1 to the input double word value (IN) and loads the result in a double word value (OUT), as is shown in the equation:

Decrement Word

Symbol:



Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

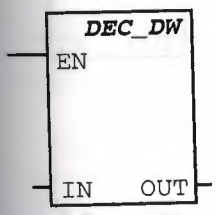
Description of operation:

The Decrement Word (DEC_W) box subtracts 1 from the input word value (IN) and loads the result in a word value (OUT), as is shown in the equation:

IN−1=OUT

Decrement Double Word

Symbol:



Operands:

IN (Dword):	VD, ID, QD, MD, SMD,AC , HC, Constant, *VD,*AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:

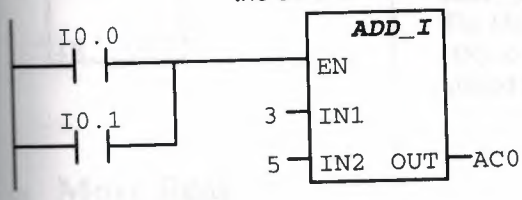
The Decrement Double Word (DEC_DW) box subtracts 1 from the input double word value (IN) and loads the result in a double word value (OUT), as is shown in the equation:

IN − 1 = OUT

Math/Inc/Dec Examples

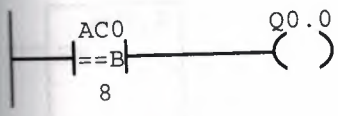
Network 1

When I0.0 or I0.1 is on then AC0 equals the sum of IN1 and IN2.



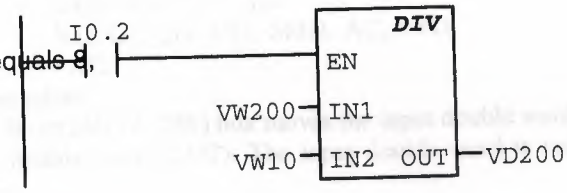
Network 2

If AC0 equals 8, turn on Q0.0 .



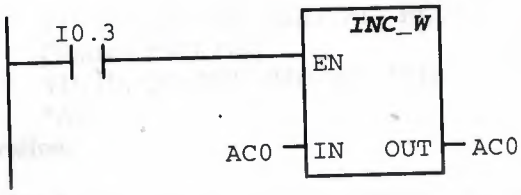
Network 3

VW200 is divided by VW10. The quotient is put in VW202, and the remainder is put in VW200. (*Note: VD200 contains VW200 and VW202.*)



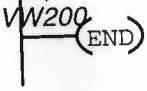
Network 4

When I0.3 is on, then the value in AC0 is incremented by 1 and stored in AC0.



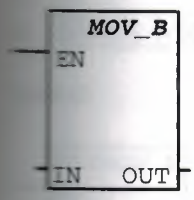
Network 5

End of the main user program.



Move Byte

Symbol:



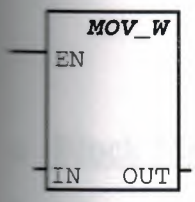
Operands:

IN (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, AC, *VD, *AC

Description of operation:
The Move Byte (MOV_B) box moves the input byte (IN) to the output byte (OUT). The input byte is not altered by the move

Move Word

Symbol:



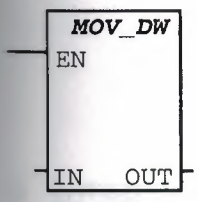
Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, AQW, *VD, *AC

Description of operation:
The Move Word (MOV_W) box moves the input word (IN) to the output word (OUT). The input word is not altered by the move.

Move Double Word

Symbol:



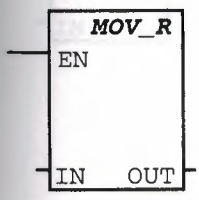
Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC, &VB, &IB, &QB, &MB, &T, &C
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:
The Move Double Word (MOV_DW) box moves the input double word (IN) to the output double word (OUT). The input double word is not altered by the move.

Move Real

Note: CPU 214 only.
Symbol:



Operands:

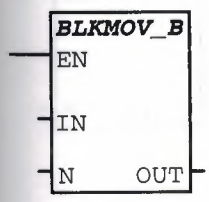
IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:

The Move Real (MOV_R) box moves a 32-bit real input double word (IN) to the output double word (OUT). The input double word is not altered by the move.

Block Move Byte

Symbol:



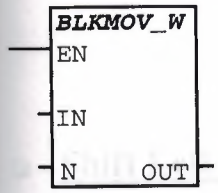
Operands:

IN (byte):	VB, IB, QB, MB, SMB *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:
The Block Move Byte (BLKMOV_B) box moves the number of bytes specified (N), from the input array starting at IN, to the output array starting at OUT.N has a range of 1 to 255.

Block Move Word

Symbol:



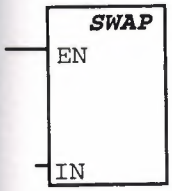
Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AIW, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AQW, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:
The Block Move Word (BLKMOV_B) box moves the number of words specified (N), from the input array starting at IN, to the output array starting at OUT. N has a range of 1 to 255.

Swap

Symbol:



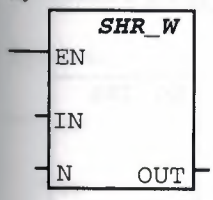
Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:
The Swap Byte box exchanges the most-significant byte with the least-significant byte of the word (IN).

❑ Shift Right Word

Symbol:



Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

The Shift Right Word (SHR_W) box shifts the word value (IN) right by the shift count (N), and loads the result in the output word (OUT).

SM1.0 (zero)= 1 if OUT = 0

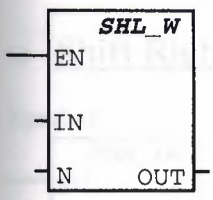
SM1.1 (overflow) = 1 if last bit shifted out = 0

Note: When IN ≠ OUT:

- If N and OUT are direct-addressed operands, and if OUT contains N, then the instruction is invalid.
- If N is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.
- If N and OUT are indirect address pointers and the pointers are equal, then the instruction is invalid.

❑ Shift Left Word

Symbol:



Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

The Shift Left Word (SHL_W) box shifts the word value (IN) left by the shift count (N), and loads the result in the output word (OUT).

SM1.0 (zero)= 1 if OUT = 0

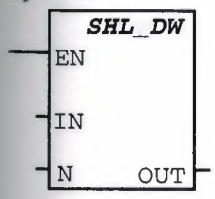
SM1.1 (overflow) = 1 if last bit shifted out = 0

Note: When IN ≠ OUT:

- If N and OUT are direct-addressed operands, and if OUT contains N, then the instruction is invalid.
- If N is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.
- If N and OUT are indirect address pointers and the pointers are equal, then the instruction is invalid.

Shift Left Double Word

Symbol:



Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:
The Shift Left Double Word (SHL_DW) box shifts the double word value (IN) left by the shift count (N), and loads the result in the output double word (OUT).

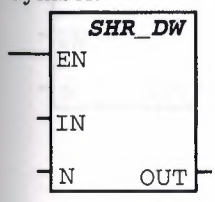
SM1.0 (zero)= 1 if OUT = 0
SM1.1 (overflow) = 1 if last bit shifted out = 0

Note: When IN ≠ OUT:

- If N and OUT are direct-addressed operands, and if OUT contains N, then the instruction is invalid.
- If N is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.
- If N and OUT are indirect address pointers and the pointers are equal, then the instruction is invalid.

Shift Right Double Word

Symbol:



Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:
The Shift Right Double Word (SHR_DW) box shifts the double word value (IN) right by the shift count (N), and loads the result in the output double word (OUT).

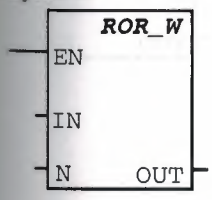
SM1.0 (zero)= 1 if OUT = 0
SM1.1 (overflow) = 1 if last bit shifted out = 0

Note: When IN ≠ OUT:

- If N and OUT are direct-addressed operands, and if OUT contains N, then the instruction is invalid.
- If N is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.
- If N and OUT are indirect address pointers and the pointers are equal, then the instruction is invalid.

Rotate Right Word

Symbol:



Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

The Rotate Right Word (ROR_W) box rotates the word value (IN) right by the shift count (N), and loads the result in the output word (OUT).

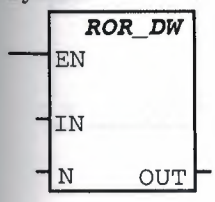
SM1.0 (zero)= 1 if OUT = 0
SM1.1 (overflow) = 1 if last bit rotated = 0

Note: When IN ≠ OUT:

- If N and OUT are direct-addressed operands, and if OUT contains N, then the instruction is invalid.
- If N is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.
- If N and OUT are indirect address pointers and the pointers are equal, then the instruction is invalid.

Rotate Right Double Word

Symbol:



Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:

The Rotate Right Double Word (ROR_DW) box rotates the double word value (IN) right by the shift count (N), and loads the result in the output double word (OUT).

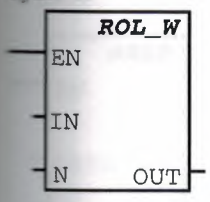
SM1.0 (zero)= 1 if OUT = 0
SM1.1 (overflow) = 1 if last bit rotated = 0

Note: When IN ≠ OUT:

- If N and OUT are direct-addressed operands, and if OUT contains N, then the instruction is invalid.
- If N is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.
- If N and OUT are indirect address pointers and the pointers are equal, then the instruction is invalid.

Rotate Left Word

Symbol:



Operands:

IN (word):	VW,T,C,IW,QW,MW,SMW, AC,AIW, Constant, *VD,*AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

The Rotate Left Word (ROL_W) box rotates the word value (IN) left by the shift count (N), and loads the result in the output word (OUT).

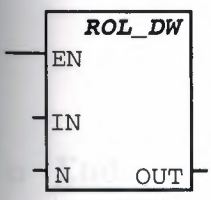
SM1.0 (zero)= 1 if OUT = 0
SM1.1 (overflow) = 1 if last bit rotated = 0

Note: When IN ≠ OUT:

- If N and OUT are direct-addressed operands, and if OUT contains N, then the instruction is invalid.
- If N is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.
- If N and OUT are indirect address pointers and the pointers are equal, then the instruction is invalid.

Rotate Left Double Word

Symbol:



Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:

The Rotate Left Double Word (ROL_DW) box rotates the double word value (IN) left by the shift count (N), and loads the result in the output double word (OUT).

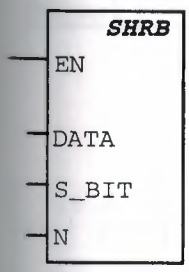
SM1.0 (zero)= 1 if OUT = 0
SM1.1 (overflow) = 1 if last bit rotated = 0

Note:
When IN ≠ OUT:

- If N and OUT are direct-addressed operands, and if OUT contains N, then the instruction is invalid.
- If N is an indirect address and OUT is a direct address containing one of the bytes of the indirect address pointer, then the instruction is invalid.
- If N and OUT are indirect address pointers and the pointers are equal, then the instruction is invalid.

Shift Register Bit

Symbol:



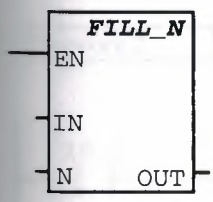
Operands:

DATA, S_BIT (bit):	I, Q, M, SM, T, C, V
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:
The Shift Register Bit (SHRB) instruction shifts the value of DATA into the shift register. S_BIT specifies the least-significant bit of the shift register. N specifies the length of the shift register and the direction of the shift (shift plus = N, shift minus = –N).

Fill Memory

Symbol:



Operands:

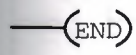
IN (word):	VW, T, C, IW, QW, MW, SMW, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AQW, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:
The Fill Memory Box (FILL_N) fills the memory starting at the output word (OUT) with the word input pattern (IN) for the number of words specified by N. N has a range of 1 to 255.

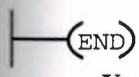
)

End

Symbols:



Conditional End



Unconditional End

Operands:
(none)
Description of operation:
The Conditional End coil terminates the main user program based on the condition of the preceding logic.
The Unconditional End coil must be used to terminate the user program.

Stop

Symbol:

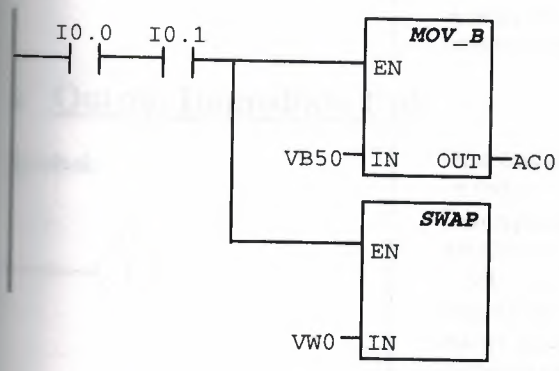
Operands:
(none)
Description of operation:

The Stop coil terminates execution of the user program by causing a transitiontothestopmode.

Move / Shift / Rotate / Fill Examples

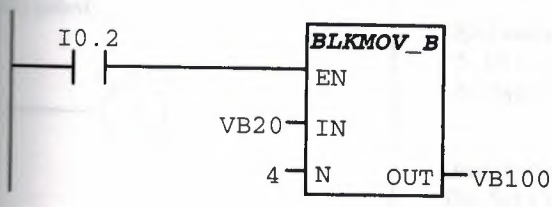
Network 1

When I0.0 and I0.1 are on then move VB50 to AC0, and swap the most significant byte (MSB) of VW0 with the LSB of VW0.



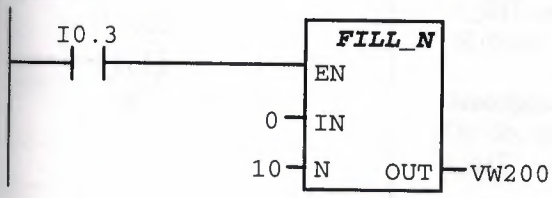
Network 2

When I0.2 is on then move VB20-VB23 toVB100-VB103.



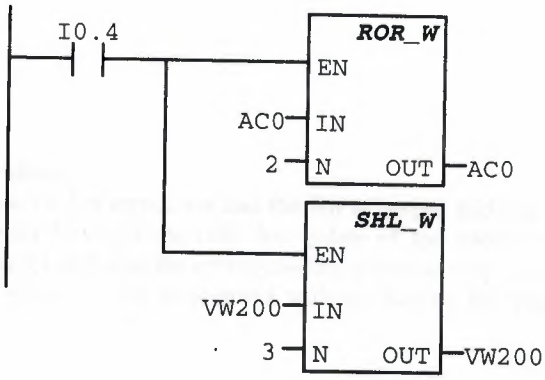
Network 3

When I0.3 is on then fill VW200-VW218 with 0's.



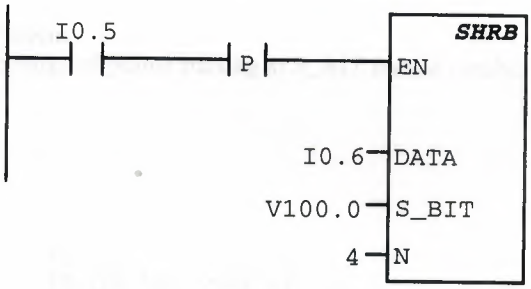
Network 4

When I0.4 is on, then the word value in AC0 is rotated right twice and stored in AC0, and the word value in VW200 is shifted left 3 times and stored in VW200.



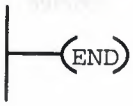
Network 5

Upon every 0 to 1 transition of I0.5, the value of I0.6 is shifted into the shift register starting at V100.0 and of length 4.



Network 6

Main end of the user program.



❑ Output

Symbol:

```
n
——( )
```

Operands:
n (bit): I, Q, M, SM, T, C, V

Description of operation:
An Output coil is turned on and the Bit stored at address ***n*** is set to 1 when power flows to the coil.
A negated output can be created by placing a NOT (Invert Power Flow) contact before an output coil.

❑ Output Immediate Coil

Symbol:

```
n
——(I)
```

Operands:
n (bit): Q

Description of operation:
An Output Immediate Coil is turned on and the Bit at output address ***n*** is set to 1 when power flows to the coil. An update of the addressed image register output Bit and also the corresponding physical output Bit occurs immediately after the coil is scanned without waiting for scan cyclecompletion.

❑ Set

Symbol:

```
S_BIT
——(S)
N
```

Operands:
S_BIT (bit): I, Q, M, SM, T, C, V
N (byte): IB, QB, MB, SMB, VB, AC, Constant, *VD, *AC

Description of operation:
The Set Coil sets the range of points starting at S_BIT for the number of pointsspecifiedbyN

❑ Set Immediate Coil

Symbol:

```
S_BIT
——(S_I)
N
```

Operands:
S_BIT (bit): Q
N (byte): IB, QB, MB, SMB, VB, AC, Constant, *VD, *AC

Description of operation:
The Set Immediate Coil immediately sets the range of points starting at S_BIT for the number of points specified by N .

❑ Reset Coil

Symbol:

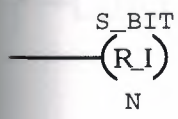
```
S_BIT
——(R)
N
```

Operands:
S_BIT (bit): I, Q, M, SM, T, C, V
N (byte): IB, QB, MB, SMB, VB, AC, Constant, *VD, *AC

Description of operation:
The Reset Coil resets the range of points starting at S_BIT for the number of points specified by N. If S_BIT is specified to be either a T or a C bit, then both the timer/counter bit and the timer/counter current valueareresset.

❑ Reset Immediate Coil

Symbol:



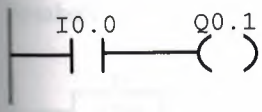
Operands:

S_BIT (bit):	Q
N (byte):	IB, QB, MB, SMB, VB, AC, Constant, *VD, *AC

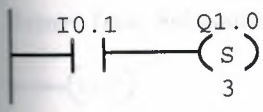
Description of operation:
The Reset Immediate Coil immediately resets the range of points starting at S_BIT for the number of points specified by N .

Ladder Output Coil Examples

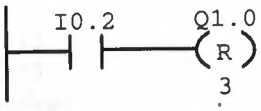
Network 1 — When I0.0 is on, then output Q0.1 is turned on.



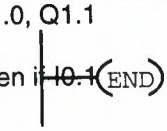
Network 2 — When I0.1 is on, then outputs Q1.0, Q1.1 and Q1.2 are set (turned on). These outputs will remain on, even if I0.1 is turned off, until they are reset.



Network 3 — When I0.2 is turned on, then outputs Q1.0, Q1.1 and Q1.2 are reset (turned of

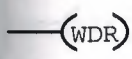


Network 4 — End of the main user program.



❑ Watchdog Reset

Symbol:

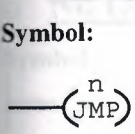


Operands:
(none)

Description of operation:
The Watchdog Reset (WDR) coil allows the watchdog timer to be retriggered. This extends the time the scan takes without getting a watchdog error.

❑ Jump

Symbol:

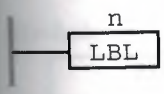


Operands:
n: CPU 212: 0-63
CPU 214: 0-255

Description of operation:
The Jump to Label (JMP) coil performs a branch to the specified label (*n*)withintheprogram

Label

Symbol:

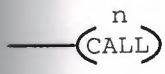


Operands:
n: CPU 212: 0-63
CPU 214: 0-255

Description of operation:
The Label (LBL) instruction marks the location of the jump destination (*n*). The CPU 212 allows 64 labels, and the CPU 214 allows 256.

Call

Symbol:

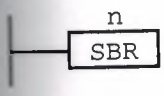


Operands:
n: CPU 212: 0-15
CPU 214: 0-63

Description of operation:
The Subroutine Call (CALL) coil transfers control to the subroutine (n).

Subroutine

Symbol:

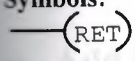


Operands:
n: CPU 212: 0-15
CPU 214: 0-63

Description of operation:
The Subroutine (SBR) label marks the beginning of the subroutine (*n*). The CPU 212 supports 16 subroutines, and the CPU 214 supports 64.

Return

Symbols:



Conditional Return from Subroutine

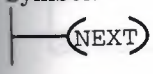
Unconditional Return from Subroutine

Operands:
(none)

Description of operation:
The Conditional Return from Subroutine coil may be used to terminate a subroutine, based on the condition of the preceding logic.
The Unconditional Return from Subroutine coil must be used to terminate each subroutine.

Next

Symbol:

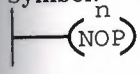


Operands:
(none)

Description of operation:
The NEXT coil marks the end of the FOR loop, and sets the top of stack to1.

No Operation

Symbol:

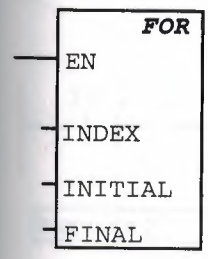


Operands:
n: 0-255

Description of operation:
The No Operation (NOP) coil has no effect on the user program execution. The operand *n* is a number from 0-255.

For

Symbol:



Operands:

INDEX (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC
INITIAL (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
FINAL (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:

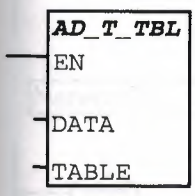
The FOR box executes the code between the FOR and the NEXT. You must specify the current loop count (INDEX), the starting value (INITIAL), and the ending value (FINAL). If the starting value is greater than the final value, the loop is not executed. After each execution of the instructions between the FOR and the NEXT instruction, the INDEX value is incremented and the result is compared to the final value. If the INDEX is greater than the final value, the loop is terminated.

For example, given an INITIAL value of 1, and a FINAL value of 10, the instructions between the FOR and the NEXT are executed 10 times with the INDEX value incrementing 1,2,3, . . 10.

Add to Table

Note: Table and Find instructions are supported by the CPU 214 only.

Symbol:



Operands:

DATA (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
TABLE (word):	VW, T, C, IW, QW, MW, SMW, *VD, *AC

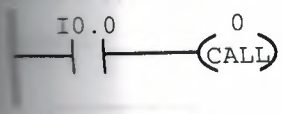
Description of operation:

The Add To Table (AD_T_TBL) box adds word values (DATA) to the table (TABLE). The first value of the table is the maximum table length (TL). The second value is the entry count (EC) that specifies the number of entries in the table. New data are added to the table after the last entry. Each time new data are added to the table, the entry count (EC) is incremented. If you try to overfill the table, the Table Full memory bit (SM1.4)isset.

Ladder Program Control Examples

Network 1

When I0.0 is on, execute Subroutine 0.

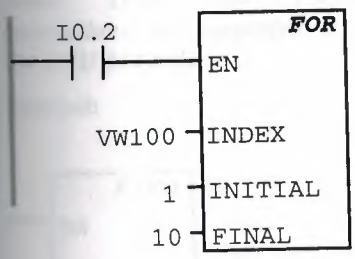


Network 2

When I0.1 is on, jump to Label 1.

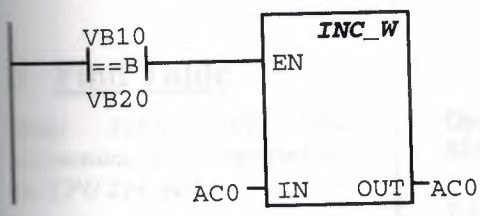
Network 3

When I0.2 is on, execute the For/Next loop 10 times.



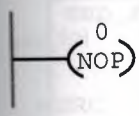
Network 4

If VB10 = VB20, then increment AC0 by 1.



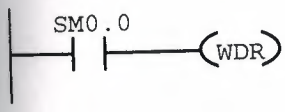
Network 5

This network does nothing.



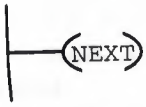
Network 6

SM0.0 is always on, therefore the Watchdog Timer is extended to allow a longer scan.



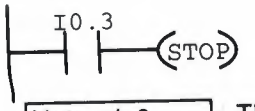
Network 7

This is the end of the For/Next loop.



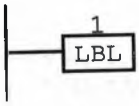
Network 8

If I0.3 comes on, then the CPU goes to Stop mode.



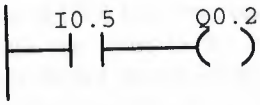
Network 9

The Jump in Network #2 jumps to this location.



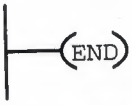
Network 10

When I0.5 is on, turn on Q0.2.



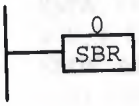
Network 11

End of the main user program.



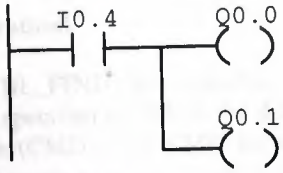
Network 12

Start of Subroutine 0.



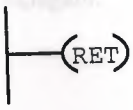
Network 13

If I0.4 is on, then turn on Q0.0 and Q0.1.



Network 14

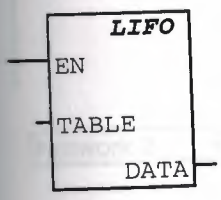
End of Subroutine 0.



LIFO (Last In First Out)

Note: Table and Find instructions are supported by the CPU 214 only.

Symbol:



Operands:

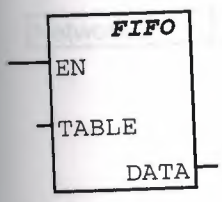
TABLE (word):	VW, T, C, IW, QW, MW, SMW, *VD, *AC
DATA (word):	VW, T, C, IW, QW, MW, SMW, AC, AQW, *VD, *AC

Description of operation:
The Last In First Out (LIFO) box removes the last entry in the table (TABLE), and outputs the value to the location (DATA). The entry count (EC) in the table is decremented for each instruction execution. If you try to remove an entry from an empty table, the Table Empty memory bit (SM1.5) is set.

FIFO (First In First Out)

Note: Table and Find instructions are supported by the CPU 214 only.

Symbol:



Operands:

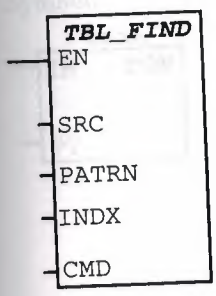
TABLE (word):	VW, T, C, IW, QW, MW, SMW, *VD, *AC
DATA (word):	VW, T, C, IW, QW, MW, SMW, AC, AQW, *VD, *AC

Description of operation:
The First In First Out (FIFO) box removes the first entry in the table (TABLE), and outputs the value to the location (DATA). All other entries of the table are shifted up one location. The entry count (EC) in the table is decremented for each instruction execution. If you try to remove an entry from an empty table, the Table Empty memory bit (SM1.5)isset.

Find Table

Note: Table and Find instructions are supported by the CPU 214 only.

Symbol:



Operands:

SRC (word):	VW, T, C, IW, QW, MW, SMW, *VD, *AC
PATRN (word):	VW, T, C, IW, QW, MW, SMW, AIW, AC, Constant, *VD, *AC
INDX (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC
CMD:	1-4

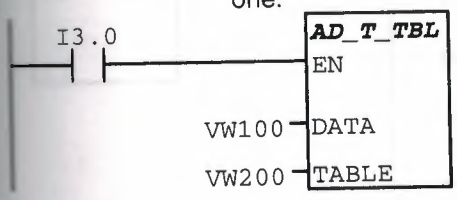
Description of operation:

The Find Table (TBL_FIND) box searches the table (SRC), starting with the table entry specified by INDX, for the data value (PATRN) that matches the criteria (CMD). The CMD parameter is given a numeric value 1-4 that corresponds to =, <>, <, and >, respectively.
If a match is found, the INDX points to the matching entry in the table. If a match is not found, the INDX has a value equal to the entry count. To find the next matching entry, the INDX must be incremented before invokingtheTBL_FINDagain.

Ladder Table / Find Instruction Examples

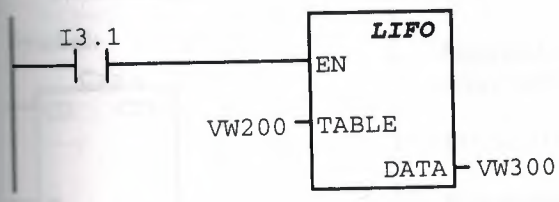
Network 1

When I3.0 is on, the value VW100 is added to the table starting at VW200. The EC (entry count) is incremented by one.



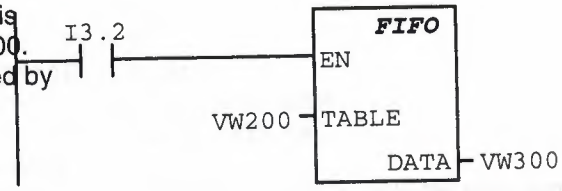
Network 2

When I3.1 is on, the last data value of the table starting at VW200 is output to the data location VW300. The EC is decremented by one.



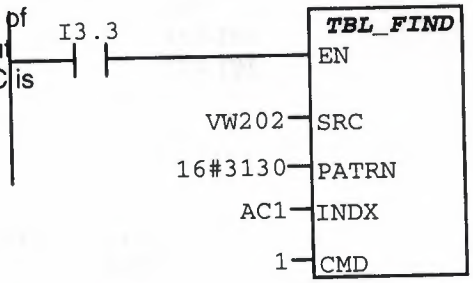
Network 3

When I3.2 is on, the first data value of the table starting at VW200 is output to the data location VW300. The EC is decremented by one.



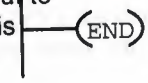
Network 4

When I3.3 is on, the table VW202 is searched for a value equal to 3130 Hex.



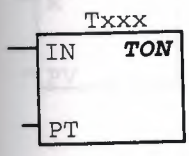
Network 5

End of the main user program.



❑ Timer – On Delay

Symbol:



Operands:

Txx (word):	CPU 212: 32-63
	CPU 214: 32-63, 96-127
PT (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

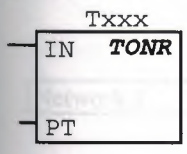
Description of operation:

The On–Delay Timer (TON) box times up to the maximum value when the enabling Input (IN) comes on. When the current value (Txxx) is >= the Preset Time (PT), the timer bit turns on. It resets when the enabling input goes off. Timing stops upon reaching the maximum value.

	CPU 212/214	CPU 214
1 ms	T32	T96
10 ms	T33-T36	T97-T100
100 ms	T37-T63	T101-T127

❑ Timer – Retentive On Delay

Symbol:



Operands:

Txxx (word):	CPU 212: 0-31 CPU 214: 0-31, 64-95
PT (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

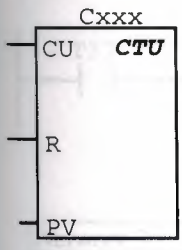
Description of operation:

The Retentive On Delay Timer (TONR) box times up to the maximum value when the enabling Input (IN) comes on. When the current value (Txxx) is >= the Preset Time (PT), the timer bit turns on. Timing stops when the enabling input goes off, or upon reaching the maximum value.

	CPU 212/214	CPU 214
1 ms	T0	T64
10 ms	T1-T4	T65-T68
100 ms	T5-T31	T69-T95

❑ Count Up

Symbol:



Operands:

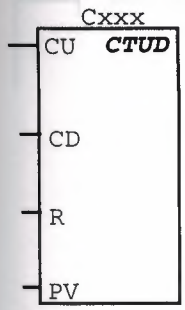
Cxxx (word):	CPU 212: 0-63 CPU 214: 0-127
PV (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:

The Count Up (CTU) box counts up to the maximum value on the rising edges of the Count Up (CU) input. When the current value (Cxxx) is >= to the Preset Value (PV), the counter bit (Cxxx) turns on. It resets when the Reset (R) input turns on. It stops counting upon reaching the maximum value (32,767).

❑ Count Up / Down

Symbol:



Operands:

Cxxx (word):	CPU 212: 0-63 CPU 214: 0-127
PV (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

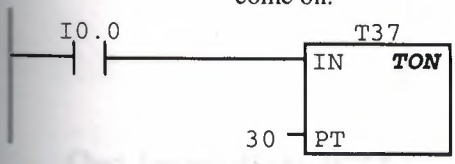
Description of operation:

The Count Up/Down (CTUD) box counts up on rising edges of the Count Up (CU) input. It counts down on the rising edges of the Count Down (CD) input. When the current value (Cxxx) is >= to the Preset Value (PV), the counter bit (Cxxx) turns on. It stops counting up upon reaching the maximum value (32,767), and stops counting down upon reaching the minimum value (-32,768). It resets when the Reset (R) inputturnson.

Ladder Timer / Counter Examples

Network 1

When I0.0 is on then the timer will start. After 3 seconds (30 X 100ms) T37 bit will come on.

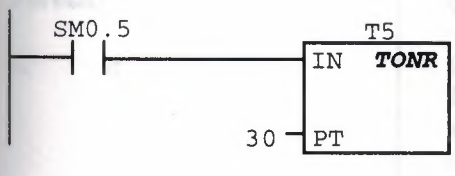


Network 2

When Timer 37 reaches its preset, turn on Q0.0 .

Network 3

When SM0.5 (1 sec. clock pulse, .5 sec. on and .5 sec. off) is ON, then the timer will time. The T5 bit will come on after 6 seconds.

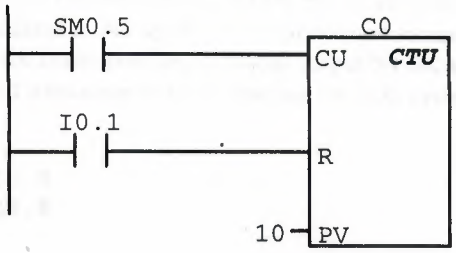


Network 4

When Timer 5 reaches its preset, turn on Q0.1 .

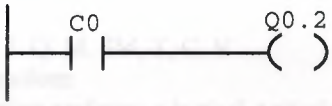
Network 5

By using SM0.5 (1 sec. clock pulse) the counter will count pulses and turn on the C0 bit when a count of 10 is reached. I0.0 resets the counter.



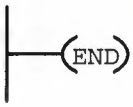
Network 6

When C0 reaches its preset, turn on Q0.2 .



Network 7

End of the main user program.



6.2: *Statement List Instruction Set:*

Out (STL)

Format:

= *n*

Operands:
n (bit): I, Q, M, SM, T, C, V

Description of operation:
The Out (=) instruction copies the bit value on the top of the logic stack to address ***n***.

Example:

```
LD    I0.0
=     Q2.0
```

Out Immediate (STL)

Format:

=I *n*

n (bit): Q

Description of operation:
The Out Immediate (=I) instruction copies the bit value on the top of the logic stack to address ***n***. An update of the addressed image register output bit and also the corresponding physical output bit occurs immediately after =I execution without waiting for scan cycle completion.

Example:

```
LDI   I0.0
=I    Q2.0
```

Operands:

And (STL)

Format:

A *n*

Operands:
n (bit): I, Q, M, SM, T, C, V

Description of operation:
The And (A) instruction performs a logical And of the bit value at address ***n*** with the top of logic stack value. The result becomes the new top of logic stack value.

Example:

```
LD    I0.1
A     I0.2
=     Q1.0
```

And Immediate (STL)

Format:

AI *n*

Operands:
n (bit): I

Description of operation:
The And Immediate (AI) instruction performs a logical And of the bit value at address ***n*** with the top of logic stack value. The result becomes the new top of stack value. A physical input read and stack operation occurs immediately after AI execution without waiting for scan cycle completion. The image register is not updated.

Example:

```
LDI   I0.1
AI    I0.2
=I    Q1.0
```

And Load (STL)

Format:
ALD

Operands:
(none)

Description of operation:
The And Load (ALD) instruction performs a logical And on the bit values in the first (top) and second levels of the logic stack. The result is loaded to the top of stack and stack depth is reduced by one.

Example:

```
LD      I0.0
LD      I0.1
LD      I2.0
A       I2.1
OLD
ALD
```

```
LD      I0.0
LPS
LD      I0.5
O       I0.6
ALD
=       Q7.0
LRD
LD      I2.1
O       I1.3
ALD
=       Q6.0
LPP
A       I1.0
=       Q3.0
```

And Not (STL)

Format:
AN *n*

Operands:
n (bit): I, Q, M, SM, T, C, V

Description of operation:
The And Not (AN) instruction performs a logical And Not of the bit value at address **n** with the top of stack value. The result becomes the new top of stack value.

Example:

```
LD      I0.1
AN      I0.2
=       Q1.0
```

And Not Immediate (STL)

Format:

ANI *n*

Operands:
n (bit): I

Description of operation:
The And Not Immediate (ANI) instruction performs a logical And Not of the bit value at address ***n*** with the top of stack value. The result becomes the new top of stack value. A physical input read and stack operation occurs immediately after ANI execution without waiting for scan cycle completion. The image register is not updated.

Example:

```
LDI     I0.1
ANI     I0.2
=I      Q1.0
```

Edge Down (STL)

Format:

ED

Operands:

(none)

Description of operation:

The Edge Down (ED) instruction detects a scan-to-scan transition from 1 to 0 in top of stack bit value. Upon detection of such a transition, the top of stack value is set to 1; otherwise it is set to 0.

Example:

```
LD    I0.2
ED
=     Q2.2
```

Edge Up (STL)

Format:

EU

(none)

Description of operation:

The Edge Up (EU) instruction detects a scan-to-scan transition from 0 to 1 in top of stack bit value. Upon detection of such a transition, the top of stack value is set to 1; otherwise it is set to 0.

Example:

```
LD    I0.1
EU
=     Q2.1
```

Load (STL)

Format:

LD *n*

Operands:

n (bit): I, Q, M, SM, T, C, V

Description of operation:

The Load (LD) instruction copies the bit value at address ***n*** to the top of the logic stack. Other stack bit values move down one level.

Example:

```
LD    I0.1
A     I0.2
=     Q1.0
```

Load Immediate (STL)

Format:

LDI *n*

Operands:

n (bit): I

Description of operation:

The Load Immediate (LDI) instruction copies the bit value at address ***n*** to the top of the logic stack immediately after execution without waiting for scan cycle completion. The image register is not updated. Other stack bit values move down one level.

Example:

```
LDI   I0.1
AI    I0.2
=I    Q1.0
```

Load Not (STL)

Format:

LDN *n*

Operands:

n (bit): I, Q, M, SM, T, C, V

Description of operation:

The Load Not (LDN) instruction copies the logical Not of the bit value at image register address ***n*** to the top of the logic stack. Other stack bit values move down one level.

Example:

```
LDN   I0.1
AN    I0.2
=     Q1.0
```

Load Not Immediate (STL)

Format:

LDNI *n*

Operands:
n (bit): I

Description of operation:
The Load Not Immediate (LDNI) instruction copies the logical Not of the bit value at address ***n*** to the top of the logic stack immediately after execution without waiting for scan cycle completion. Other stack bit values move down one level.

Example:

```
LDNI  I0.1
ANI   I0.2
=I    Q1.0
```

Logic Pop (STL)

Format:

LPP

Operands:
(none)

Description of operation:
The Logic Pop (LPP) instruction pops one value off of the stack. The second level bit value becomes the new top of stack value. Other stack bit values move up one level.

Example:

```
LD    I0.0
LPS
LD    I0.5
O     I0.6
ALD
=     Q7.0
LRD
LD    I2.1
O     I1.3
ALD
=     Q6.0
LPP
A     I1.0
=     Q3.0
```

Logic Push (STL)

Format:

LPS

Operands:
(none)

Description of operation:
The Logic Push (LPS) instruction duplicates the top of stack bit value and pushes this value onto the stack. The bottom of the stack is pushed off and lost.

Example:

```
LD    I0.0
LPS
LD    I0.5
O     I0.6
ALD
=     Q7.0
LRD
LD    I2.1
O     I1.3
ALD
=     Q6.0
LPP
A     I1.0
=     Q3.0
```

Logic Read (STL)

Format:

LRD

Operands:
(none)
Description of operation:
The Logic Read (LRD) instruction copies the second stack value to the top of stack. The stack is not pushed or popped, but the old top of stack value is destroyed by the copy.
Example:

```
LD      I0.0
LPS
LD      I0.5
O       I0.6
ALD
=       Q7.0
LRD
LD      I2.1
O       I1.3
ALD
=       Q6.0
LPP
A       I1.0
=       Q3.0
```

Logical Negation (STL)

Format:

NOT

Operands:
(none)
Description of operation:
The Logical Negation (NOT) instruction changes the top of stack bit value from 0 to 1, or from 1 to 0.
Example:

```
LD      I0.0
NOT
=       Q2.0
```

Or (STL)

Format:

O *n*

Operands:
n (bit): I, Q, M, SM, T, C, V
Description of operation:
The Or (O) instruction performs a logical Or of the bit value at address ***n*** with the top of logic stack value. The result becomes the new top of stack value.
Example:

```
LD      I1.1
O       I1.2
=       Q1.1
```

Or Immediate (STL)

Format:

OI *n*

Operands:

n (bit): I

Description of operation:

The Or Immediate (OI) instruction performs a logical Or of the bit value at input module address ***n*** with the top of stack value. The result becomes the new top of stack value. A physical input read and stack operation occurs immediately after OI execution without waiting for scan cycle completion. The image register is not updated.

Example:

```
LDI   I1.1
OI    I1.2
=I    Q1.1
```

Or Load (STL)

Format:

OLD

Operands:

(none)

Description of operation:

The Or Load (OLD) instruction performs a logical Or with the bit values in the first (top) and second levels of the stack. The result is loaded to the top of stack. After execution of OLD, stack depth is reduced by one.

Example:

```
LD    I0.0
LD    I0.1
LD    I2.0
A     I2.1
OLD
ALD
```

Or Not (STL)

Format:

ON *n*

Operands:

n (bit): I, Q, M, SM, T, C, V

Description of operation:

The Or Not (ON) instruction performs a logical Or Not of the bit value at address ***n*** with the top of logic stack value.

Or Not Immediate (STL)

Format:

ONI *n*

Operands:

n (bit): I

Description of operation:

The Or Not Immediate (ONI) instruction immediately performs a logical Or Not of the bit value at physical input address ***n*** with the top of logic stack value. The result becomes the new top of stack value. A physical input read and stack operation occurs immediately after ONI execution without waiting for scan cycle completion.

Example:

```
LDI   I1.1
ONI   I1.2
=I    Q1.1
```

Reset (STL)

Format:

R *S_BIT, N*

Operands:

S_BIT (bit):	I, Q, M, SM, T, C, V
N (byte):	IB, QB, MB, SMB, VB, AC, Constant, *VD, *AC

Description of operation:
The Reset (R) instruction resets a range of bit values. Bit values of 0 are written to a range starting at address S_BIT for the number of bits specified by N. If S_BIT is specified to be either a T or a C bit, then both the timer/counter bit and the timer/counter current value are reset to 0.

Example:

```
LD    I0.0
=     Q2.0
S     Q2.1, 1
R     Q2.2, 1
R     Q1.0, 3
```

Reset Immediate (STL)

Format:

RI *S_BIT, N*

Operands:

S_BIT (bit):	Q
N (byte):	IB, QB, MB, SMB, VB, AC, Constant, *VD, *AC

Description of operation:
The Reset Immediate (RI) instruction immediately resets a range of bit values. Bit values of 0 are written to a range starting at S_BIT for the number of bits specified by N. Specified bits in the image register and corresponding physical outputs are updated at execution time without waiting for scan cycle completion.

Example:

```
LDI   I0.0
=I    Q2.0
SI    Q2.1, 1
RI    Q2.2, 1
RI    Q1.0, 3
```

Set (STL)

Format:

S *S_BIT, N*

Operands:

S_BIT (bit):	I, Q, M, SM, T, C, V
N (byte):	IB, QB, MB, SMB, VB, AC, Constant, *VD, *AC

Description of operation:
The Set (S) instruction sets a range of bit values. Bit values of 1 are written to a range starting at address S_BIT for the number of bits specified by N.

Set Immediate (STL)

Format:

SI *S_BIT, N*

Operands:

S_BIT (bit): Q

N (byte): IB, QB, MB, SMB, VB, AC, Constant, *VD, *AC

Description of operation:

The Set Immediate (SI) instruction immediately resets a range of bit values. A bit value of 1 is written to a range starting at S_BIT for the number of bits specified by N. Specified bits in the image register and physical output modules are updated at execution time without waiting for scan cycle completion.

Example:

```
LDI   I0.0
=I    Q2.0
SI    Q2.1, 1
RI    Q2.2, 1
```

Read Time of Day (STL)

Note: Real Time Clock instructions are supported by the CPU 214 only.

Format:

TODR *T*

Operands:

T (byte): VB, IB, QB, MB, SMB, *VD, *AC

Description of STL operation:

Read Time of Day (TODR) reads the current date and time from the Real Time Clock. The 8 bytes of time data are written to memory with the area and starting address specified by ***T***.

Year/Month	yymm	yy - 0 to 99	mm - 1 to 12
Day/Hour	ddhh	dd - 1 to 31	hh - 0 to 23
Minute/Second	mmss	mm - 0 to 59	ss - 0 to 59
Day of week	000d	d - 1 to 7 d - 0	1 = Sunday Day of week remains 0

Example:

```
LD     I2.1          //Enable READ_RTC
TODR   VB400         //Read clock
MOVB   VB400, AC0    //Move year value
                     //to accumulator
```

Example Memory Data Starting at VB400:

Note: The time of day clock initializes the following date and time after extended power outages or memory has been lost:

Date:	01-Jan-90
Time:	00:00:00
Day of Week	Sunday

Note: Do not use the TODR/TODW instructions in both the main program and in an interrupt routine. If you do this and the TOD instruction is executing when the the interrupt that also executes the TOD instruction occurs, then the TOD instruction in the interrupt routine is not executed. SM4.5 is then set, indicating that two simultaneous accesses to the clock were attempted.

Write Time of Day (STL)

Note: Real Time Clock instructions are supported by the CPU 214 only.

Format:

TODW *T*

Operands:

T (byte): VB, IB, QB, MB, SMB, *VD, *AC

Description of STL operation:

Write Time of Day (TODW) sets a date and time into the Real Time Clock. The 8 bytes of time data are read from a memory area with the starting address specified by ***T***.

The Date and Time setting data must be in BCD format (4 bits per digit; decimal digits 0-9 only) and previously stored in the specified memory location before execution of TODW.

Year/Month	yymm	yy - 0 to 99	mm - 1 to 12
Day/Hour	ddhh	dd - 1 to 31	hh - 0 to 23
Min/Sec	mmss	mm - 0 to 59	ss - 0 to 59
Day of week	000d	d - 1 to 7 d - 0	1 = Sunday Day of week remains 0

Compare Byte Equal Instructions (STL)

Format:

```
LDB=  n1, n2
AB=   n1, n2
OB=   n1, n2
```

Operands:

n1, n2 (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:
The **Load Byte (LDB)**, **And Byte (AB)**, and **Or Byte (OB) Compare Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 = n2.

Example:

```
LD    Q0.0
AB=   VB4, VB8
=     Q2.0
```

Compare Byte Greater Than or Equal Instructions (STL)

Format:

```
LDB>=  n1, n2
AB>=   n1, n2
OB>=   n1, n2
```

Operands:

n1, n2 (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:
The **Load Byte (LDB), And Byte (AB),** and **Or Byte (OB) Compare Greater Than or Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 ≥ n2.

Example:

```
LD    Q0.0
AB>=  VB4, VB8
=     Q2.0
```

Compare Byte Less Than or Equal Instructions (STL)

Format:

```
LDB<=  n1, n2
AB<=   n1, n2
OB<=   n1, n2
```

Operands:

n1, n2 (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:
The **Load Byte (LDB), And Byte (AB),** and **Or Byte (OB) Compare Less Than or Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 ≤ n2.

Example:

```
LD    Q0.0
AB<=  VB4, VB8
=     Q2.0
```

Compare Word Equal Instructions (STL)

Format:

```
LDW=  n1, n2
AW=   n1, n2
OW=   n1, n2
```

Operands:

n1, n2 (word):	VW, T, C, IW, QW, MW, SMW, AC, AI Constant, *VD, *AC

Description of operation:

The **Load Word (LDW), And Word (AW),** and **Or Word (OW) Compare Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 = n2.

Compare Word Greater Than or Equal Instructions (STL)

Format:

```
LDW>=   n1, n2
AW>=    n1, n2
OW>=    n1, n2
```

Operands:

n1, n2 (word): VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:

The **Load Word (LDW), And Word (AW),** and **Or Word (OW) Compare Greater Than or Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 ≥ n2.

Example:

```
LD      Q0.0
AW>=    VW4, VW8
=       Q2.0
```

Compare Word Less Than or Equal Instructions (STL)

Format:

```
LDW<=   n1, n2
AW<=    n1, n2
OW<=    n1, n2
```

Operands:

n1, n2 (word): VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:

The **Load Word (LDW), And Word (AW),** and **Or Word (OW) Compare Less Than or Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 ≤ n2.

Example:

```
LD      Q0.0
AW<=    VW4, VW8
=       Q2.0
```

Compare Double Word Equal Instructions (STL)

Format:

```
LDD=    n1, n2
AD=     n1, n2
OD=     n1, n2
```

Operands:

n1, n2 (Dword): VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC

Description of operation:

The **Load Double Word (LDD), And Double Word (AD),** and **Or Double Word (OD) Compare Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 = n2.

Example:

```
LD      Q0.0
OD=     VD6, VD20
=       Q2.0
```

Compare Double Word Greater Than or Equal Instructions (STL)

Format:

```
LDD>=   n1, n2
AD>=    n1, n2
OD>=    n1, n2
```

Operands:

n1, n2 (Dword): VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC

Description of operation:

The **Load Double Word (LDD), And Double Word (AD),** and **Or Double Word (OD) Compare Greater Than or Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 ≥ n2.

Example:

```
LD      Q0.0
OD>=    VD6, VD20
=       Q2.0
```

Compare Double Word Less Than or Equal Instructions (STL)

Format:

```
LDD<=   n1, n2
AD<=    n1, n2
OD<=    n1, n2
```

Operands:

n1, n2 (Dword): VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC

Description of operation:

The **Load Double Word (LDD), And Double Word (AD),** and **Or Double Word (OD) Compare Less Than or Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 ≤ n2.

Example:

```
LD      Q0.0
OD<=    VD6, VD20
=       Q2.0
```

Compare Real Equal Instructions (STL)

Note: Compare Real instructions are supported by the CPU 214 only.

Format:

```
LDR=    n1, n2
AR=     n1, n2
OR=     n1, n2
```

Operands:

n1, n2 (Dword): VD, ID, QD, MD, SMD, SD, AC, HC, Constant, *VD, *AC

Description of operation:

The **Load Real (LDR), And Real (AR),** and **Or Real (OR) Compare Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 = n2.

Example:

```
LD      Q0.0
OR=     VD6, VD20
=       Q2.0
```

Compare Real Greater Than or Equal Instructions (STL)

Note: Compare Real instructions are supported by the CPU 214 only.

Format:

```
LDR>=   n1, n2
AR>=    n1, n2
OR>=    n1, n2
```

Operands:

n1, n2 (Dword): VD, ID, QD, MD, SMD, SD, AC, HC, Constant, *VD, *AC

Description of operation:

The **Load Real (LDR), And Real (AR),** and **Or Real (OR) Compare Greater Than or Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 ≥ n2.

Example:

```
LD      Q0.0
OR>=    VD6, VD20
=       Q2.0
```

Compare Real Less Than or Equal Instructions (STL)

Note: Compare Real instructions are supported by the CPU 214 only.

Format:

```
LDR<=   n1, n2
AR<=    n1, n2
OR<=    n1, n2
```

Operands:

n1, n2 (Dword): VD, ID, QD, MD, SMD, SD, AC, HC, Constant, *VD, *AC

Description of operation:

The **Load Real (LDR), And Real (AR),** and **Or Real (OR) Compare Less Than or Equal** instructions Load, And, or Or a 1 with the top of the stack when n1 ≤ n2.

Example:

```
LD      Q0.0
OR<=    VD6, VD20
=       Q2.0
```

ASCII to Hex (STL)

Format:

ATH *IN, OUT, LEN*

Operands:

IN (byte):	VB, IB, QB, MB, SMB, *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, *VD, *AC
LEN (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:
The ASCII to HEX (ATH) instruction converts the ASCII string of length LEN, starting with the character IN, to hexadecimal digits starting at the location OUT. The maximum length of the ASCII string is 255 characters.
Legal ASCII characters are the hexadecimal values 30-39, and 41-46. If an illegal ASCII character is encountered, the conversion is terminated, and the NOT_ASCII memory bit (SM1.7) is set.

Example:

```
LD    I3.2
ATH   VB30, VB40, 3
```

Convert BCD to Integer (STL)

Format:

BCDI *IN*

Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:
The Convert BCD to Integer (BCDI) instruction converts the BCD value (IN) to an integer value. The result replaces the original input value. If the input value contains an invalid BCD digit, the BCD/BIN memory bit (SM1.6) is set.

Example:

```
LD    I3.0
BCDI  AC0
```

Decode (STL)

Format:

DECO *IN, OUT*

Operands:

IN (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:
The Decode (DECO) instruction sets the bit in the output word (OUT) that corresponds to the bit number represented by the least-significant nibble (LSN) of the input byte (IN). All other bits of the output word are set to 0.

Example:

```
LD    I3.1
DECO  AC2, VW40
```

Encode (STL)

Format:

ENCO *IN, OUT*

Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, AC, *VD, *AC

❑ Integer Double Word to Real(STL)

Note: Real Conversion instructions are supported by the CPU 214 only.

Format:

DTR *IN, OUT*

Operands:

IN (Dword):	VD, ID, QD, MD, SMD, SD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, SD, AC, *VD, *AC

Description of operation:
The Integer Double Word to Real (DTR) instruction converts a 32-bit signed integer (IN) into a 32bit real number (OUT).

Example:

```
LD    I3.1
DTR   AC1, VD40
```

❑ Segment (STL)

Format:

SEG *IN, OUT*

Operands:

IN (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, AC, *VD, *AC

Description of operation:
The Segment (SEG) instruction generates a bit pattern (OUT) that illuminates the segments of a seven-segment display. The illuminated segments represent the character in the least-significant digit of the input byte (IN).

Example:

```
LD    I3.1
SEG   VB48, AC1
```

❑ Hex to ASCII (STL)

Format:

HTA *IN, OUT, LEN*

Operands:

IN (byte):	VB, IB, QB, MB, SMB, *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, *VD, *AC
LEN (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:
The HEX to ASCII (HTA) instruction converts the hexadecimal digits, starting with the input byte IN, to an ASCII string starting at the location OUT. The number of hexadecimal digits to be converted is specified by length LEN. The maximum number of the hexadecimal digits that can be converted is 255.

Example:

```
LD    I3.2
HTA   VB30, VB40, 3
```

❑ Convert Integer to BCD (STL)

Note: CPU 214 only.

Format:

IBCD *IN*

Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:
The Convert Integer to BCD (IBCD) instruction converts the integer value (IN) to a BCD value (OUT). The result replaces the original input value. If the conversion produces a BCD number greater than 9999, the BCD/BIN memory bit (SM1.6) is set.

Truncate (STL)

Note: CPU 214 only.

Format:

TRUNC *IN, OUT*

Operands:

IN (Dword):	VD, ID, QD, MD, SMD, SD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, SD, AC, *VD, *AC

Description of operation:
The Truncate (TRUNC) instruction converts a 32-bit real number (IN) into a 32-bit signed integer (OUT). Only the whole-number portion of the real number is converted.

Example:

```
LD      I3.1
TRUNC   AC1, VD40
```

Count Up (STL)

Format:

CTU Cxxx, PV

Operands:

Cxxx (word):	CPU 212: 0-63 CPU 214: 0-127
PV (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:
The Count Up (CTU) instruction counts up to the maximum value on the rising edges of the Count Up (CU) input (the value loaded in the second stack location). The counter resets when the reset input turns on. The reset input is the top of stack value. When the current value (Cxxx) is >= to the Preset Value (PV), the counter bit (Cxxx) turns on. The counter stops counting upon reaching the maximum value (32,767).

Example:

```
LD      I4.0   //Count up input
LD      I2.0   //Reset input
CTU     48, 4
```

Count Up/Down (STL)

Format:

CTUD Cxxx, PV

Operands:

Cxxx (word):	CPU 212: 0-63 CPU 214: 0-127
PV (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:
The Count Up/Down (CTUD) instruction counts up on rising edges of the count-up input. The count-up input is the value loaded in the third stack location. The counter counts down on the rising edges of the count-down input. The count-down input is the value loaded in the second stack location. The counter resets when the reset input turns on. The reset input is the top of stack value or the first stack location. When the current value (Cxxx) is >= to the Preset Value (PV), the counter bit (Cxxx) turns on. The counter stops counting up upon reaching the maximum value (32,767), and stops counting down upon reaching the minimum value (-32,768).

Example:

```
LD      I4.0   //Count Up Clock
LD      I3.0   //Count Down Clock
LD      I2.0   //Reset
CTUD    C48, 4
```

Attach Interrupt (STL)

Format:

```
ATCH  INT, EVENT
```

Operands:

INT (byte):	CPU 212: 0-31
	CPU 214: 0-127
EVENT (byte):	CPU 212: 0, 1, 8-10, 12
	CPU 214: 0-20

Description of operation:
The Attach Interrupt (ATCH) instruction associates an interrupt event (EVENT) with an interrupt routine number (INT), and enables the interrupt event.

Example:

```
LD    SM0.1
ATCH  4, 0
ENI
```

Detach Interrupt (STL)

Format:

```
DTCH  EVENT
```

Operands:

EVENT (byte):	CPU 212: 0, 1, 8-10, 12
	CPU 214: 0-20

Description of operation:
The Detach Interrupt (DTCH) instruction dissociates an interrupt event (EVENT) from all interrupt routines, and disables the interrupt event.

Example:

```
LD    SM5.0
DTCH  0
```

Interrupt Routine (STL)

Format:

```
INT  n
```

Operands:

n (word):	CPU 212: 0-31
	CPU 214: 0-127

Description of operation:
The Interrupt Routine (INT) instruction marks the beginning of the interrupt routine (n). The maximum number of interrupts supported by the CPU 212 is 32, and by the CPU 214, 128.

Example:

```
INT  4
```

Enable Interrupt (STL)

Format:

```
ENI
```

Operands:
(*None*)

Description of operation:
The Enable Interrupt (ENI) instruction globally enables processing of all attached interrupt events.

Example:

```
LD    SM0.1
ATCH  4, 0
ENI
```

Disable Interrupt (STL)

Format:

```
DISI
```

Operands:
(*None*)

Description of operation:
The Disable Interrupt (DISI) instruction globally disables processing of all interrupt events.

Example:

```
LD    M5.0
DISI
```

Conditional Return from Interrupt (STL)

Format:

```
CRETI
```

Operands:
(*None*)
Description of operation:
The Conditional Return from Interrupt (CRETI) instruction may be used to return from an interrupt, based upon the condition of the preceding logic.
Example:

```
LD      SM5.0
CRETI
```

Return from Interrupt (STL)

Format:

```
RETI
```

Operands:
(*None*)
Description of operation:
The Return from Interrupt (RETI) instruction is an unconditional return and must be used to terminate each interrupt routine.
Example:

```
LD      SM5.0
CRETI
RETI
```

High-speed Counter Definition (STL)

Format:

```
HDEF  HSC, MODE
```

Operands:

HSC (byte):	CPU 212: 0
	CPU 214: 0-2
MODE (byte):	CPU 212: 0
	CPU 214: 0 (HSC0), 0-11 (HSC1-2)

Description of operation:
The High-speed Counter Definition (HDEF) instruction assigns a MODE to the referenced high-speed counter (HSC). Only one HDEF box may be used per counter.
Example:

```
LD      SM0.0
MOVB    16#F8, SMB47
HDEF    1, 11
MOVD    0, SMD48
MOVD    50, SMD52
ATCH    0, 13
ENI
HSC     1
```

High-speed Counter (STL)

Format:

```
HSC    N
```

Operands:

N (word):	CPU 212: 0
	CPU 214: 0-2

Description of operation:
The High-speed Counter (HSC) instruction invokes the operation defined by the special memory bit for the referenced high-speed counter. The parameter *N* specifies the high-speed counter number.
Example:

```
LD      SM0.0
MOVB    16#F8, SMB47
HDEF    1, 11
MOVD    0, SMD48
MOVD    50, SMD52
ATCH    0, 13
ENI
HSC     1
```

Pulse (STL)

Format:

```
PLS    x
```

Operands:

x (word): CPU 214: 0-1

Description of operation:

The Pulse (PLS) instruction examines the special memory bits for that pulse output (*x*). The pulse operation defined by the special memory bits is then invoked.

Example:

```
LD     SM0.0
MOVB   16#85, SMB67
MOVW   500, SMW68
MOVD   4, SMD72
ATCH   3, 19
ENI
PLS    0
```

Transmit (STL)

Format:

```
XMT    TABLE, PORT
```

Operands:

TABLE (byte): VB, IB, QB, MB, SMB, *VD, *AC

PORT (byte): 0

Description of operation:

The Transmit (XMT) instruction invokes the transmission of the data buffer (TABLE). The first entry in the data buffer specifies the number of bytes to be transmitted. PORT specifies the communication port to be used for transmission. It must always be 0.

Example:

```
LD     M6.3
A      SM4.5
XMT    *VD100, 0
```

Add Integer (STL)

Format:

```
+I     IN1, IN2
```

Operands:

IN1 (word): VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

IN2 (word): VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

The Add Integer (+I) instruction adds two 16-bit integers (IN1, IN2), and produces a 16-bit result (IN2), as is shown in the equation:

IN1 + IN2 = IN2

Example:

```
LD     I4.0
+I     AC1, AC0
MUL    AC1, VD100
DIV    VW10, VD200
```

Subtract Integer (STL)

Format:

```
-I     IN1, IN2
```

Operands:

IN1 (word): VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

IN2 (word): VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

The Subtract Integer (-I) instruction subtracts two 16-bit integers (IN1, IN2), and produces a 16-bit result (IN2), as is shown in the equation:

IN2 - IN1 = IN2

Add Double Integer (STL)

Format:

+D IN1, IN2

Operands:

IN1 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
IN2 (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:

The Add Double Integer (+D) instruction adds two 32-bit integers (IN1, IN2), and produces a 32bit result (IN2), as is shown in the equation:

IN1 + IN2 = IN2

Example:

```
LD    I4.0
+D    AC1, AC0
MUL   AC1, VD100
DIV   VW10, VD200
```

Subtract Double Integer (STL)

Format:

-D IN1, IN2

Operands:

IN1 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
IN2 (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:

The Subtract Double Integer (-D) instruction subtracts two 32-bit integers (IN1, IN2), and produces a 32-bit result (IN2), as is shown in the equation:

IN2 - IN1 = IN2

Example:

```
LD    I4.0
-D    AC1, AC0
MUL   AC1, VD100
DIV   VW10, VD200
```

Add Real (STL)

Note: CPU 214 only.

Format:

+R IN1, IN2

Operands:

IN1 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
IN2 (Dword):	VD, ID, QD, SMD, AC, *VD, *AC

Description of operation:

The Add Real (+R) instruction adds two 32-bit real numbers (IN1, IN2), and produces a 32-bit real number result (IN2), as is shown in the equation:

IN1 + IN2 = IN2

Example:

```
LD    I4.0
+R    AC1, AC0
MUL   AC1, VD100
DIV   VW10, VD200
```

Subtract Real (STL)

Note: CPU 214 only.

Format:

```
-R    IN1, IN2
```

Operands:

IN1 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
IN2 (Dword):	VD, ID, QD, SMD, AC, *VD, *AC

Description of operation:
The Subtract Real (-R) instruction subtracts two 32-bit real numbers (IN1, IN2), and produces a 32-bit real number result (IN2), as is shown in the equation:

Multiply Real (STL)

Note: CPU 214 only.

Format:

```
*R    IN1, IN2
```

Operands:

IN1 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
IN2 (Dword):	VD, ID, QD, SMD, AC, *VD, *AC

Description of operation:
The Multiply Real (*R) instruction multiplies two 32-bit real numbers (IN1, IN2), and produces a 32-bit real number product (IN2), as is shown in the equation:
IN1 * IN2 = IN2

Example:

```
LD    I4.0
*R    AC1, AC0
MUL   AC1, VD100
DIV   VW10, VD200
```

Divide Real (STL)

Note: CPU 214 only.

Format:

```
/R    IN1, IN2
```

Operands:

IN1 (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
IN2 (Dword):	VD, ID, QD, SMD, AC, *VD, *AC

Description of operation:
The Divide Real (/R) instruction divides two 32-bit real numbers (IN1, IN2), and produces a 32-bit real number quotient (IN2), as is shown in the equation:
IN2 / IN1 = IN2

Example:

```
LD    I4.0
/R    AC1, AC0
MUL   AC1, VD100
DIV   VW10, VD200
```

Multiply Integer (STL)

Format:

```
MUL   IN1, IN2
```

Operands:

IN1 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
IN2 (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:
The Multiply Integer (MUL) instruction multiplies a 16-bit integer (IN1) by the least-significant 16 bits of a 32-bit integer (IN2) and produces a 32-bit result (IN2), as is shown in the equation:
IN1 * IN2 = IN2

Example:

```
LD    I4.0
+D    AC1, AC0
MUL   AC1, VD100
DIV   VW10, VD200
```

Divide Integer (STL)

Format:

```
DIV   IN1, IN2
```

Operands:

IN1 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
IN2 (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:
The Divide Integer (DIV) instruction divides a 16-bit integer (IN1) into the least-significant 16 bits of a 32-bit integer (IN2) and produces a 32-bit result (IN2) composed of a 16-bit quotient (least significant) and a 16-bit remainder (most significant), as is shown in the equation:

Square Root (STL)

Note: CPU 214 only.

Format:

```
SQRT  IN, OUT
```

Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:
The Square Root (SQRT) instruction takes the square root of a 32-bit real number (IN) and produces a 32-bit real number result (OUT), as is shown in the equation:

Example:

```
LD    I4.0
SQRT  AC1, AC0
MUL   AC1, VD100
DIV   VW10, VD200
```

Block Move Byte (STL)

Format:

```
BMB   IN, OUT, N
```

Operands:

IN (byte):	VB, IB, QB, MB, SMB, *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:
The Block Move Byte (BMB) instruction moves the number of bytes specified (N) from the input array starting at IN to the output array starting at OUT. N has a range of 1 to 255.

Example:

```
LD    I2.1
BMB   VB20, VB100, 4
FILL  0, VW200, 10
```

Block Move Word (STL)

Format:

```
BMW   IN, OUT, N
```

Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AIW, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AQW, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:
The Block Move Word (BMW) instruction moves the number of words specified (N) from the input array starting at IN to the output array starting at OUT. N has a range of 1 to 255.

Example:

```
LD    I2.1
BMW   VW20, VW100, 4
FILL  0, VW200, 10
```

Memory Fill (STL)

Format:

```
FILL  IN, OUT, N
```

Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AIW, Constant, *VD, *AC
OUT (word):	VW, T, C, IW, QW, MW, SMW, AQW, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:
The Memory Fill (FILL) instruction fills the memory starting at the output word (OUT) with the word input pattern (IN) for the number of words specified by N. N has a range of 1 to 255.

Move Byte (STL)

Format:

```
MOVB  IN, OUT
```

Operands:

IN (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC
OUT (byte):	VB, IB, QB, MB, SMB, AC, *VD, *AC

Description of operation:
The Move Byte (MOVB) instruction moves the input byte (IN) to the output byte (OUT). The input byte is not altered by the move.
Example:

```
LD    I2.1
MOVB  VB50, AC0
SWAP  AC0
```

Move Double Word (STL)

Format:

```
MOVD  IN, OUT
```

Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC, &VB, &IB, &QB, &MB, &T, &C
OUT (Dword):	OUT: VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:
The Move Double Word (MOVD) instruction moves the input double word (IN) to the output double word (OUT). The input double word is not altered by the move.
Example:

```
LD    I2.1
MOVD  VD50, AC0
SWAP  AC0
```

Move Real (STL)

Note: CPU 214 only.

Format:

```
MOVR  IN, OUT
```

Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC
OUT (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of operation:
The Move Real (MOVR) instruction moves a 32-bit real input double word (IN) to the output double word (OUT). The input double word is not altered by the move.
Example:

```
LD    I2.1
MOVR  VD50, AC0
SWAP  AC0
```

Move Word (STL)

Format:

```
MOVW IN, OUT
```

Operands:

IN (word): VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

OUT (word): VW, T, C, IW, QW, MW, SMW, AC, AQW, *VD, *AC

Description of operation:

The Move Word (MOVW) instruction moves the input word (IN) to the output word (OUT). The input word is not altered by the move.

Example:

```
LD   I2.1
MOVW VW50, AC0
SWAP AC0
```

Swap Bytes (STL)

Format:

```
SWAP IN
```

Operands:

IN (word): VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

The Swap Bytes (SWAP) instruction exchanges the most-significant byte with the least-significant byte of the word (IN).

Example:

```
LD   I2.1
MOVR VD50, AC0
SWAP AC0
```

Network Read (STL)

Note: Network instructions are supported by the CPU 214 only.

Format:

```
NETR t, p
```

Operands:

t: VB, MB, *VD, *AC

p: Constant (CPU 214: 0)

Description of operation:

The Network Read (NETR) instruction initiates a communication operation to gather data from a remote device through the specified port (p), as defined in the description table (t). The format of the description table is CPU-specific.

You can use the NETR instruction to read up to 16 bytes of information from a remote station, and use the NETW instruction to write up to 16 bytes of information to a remote station. A maximum of eight NETR and NETW instructions may be activated at any one time. For example, you can have four NETR and four NETW instructions, or two NETR and six NETW instructions.

Example:

```
LDN  SM0.1
AN   V200.6
AN   V200.5
MOVB 2, VB201

MOVD &VB100, VD202
MOVB 3, VB206
```

Network Write (STL)

Note: Network instructions are supported by the CPU 214 only.

Format:

```
NETW t, p
```

Operands:

t: VB, MB, *VD, *AC
p: Constant
(CPU 214: 0)

Description of operation:
The Network Write (NETW) instruction initiates a communication operation to write data to a remote device through the specified port (p), as defined in the description table (t).
You can use the NETR instruction to read up to 16 bytes of information from a remote station, and use the NETW instruction to write up to 16 bytes of information to a remote station. A maximum of eight NETR and NETW instructions may be activated at any one time. For example, you can have four NETR and four NETW instructions, or two NETR and six NETW instructions.

Example:

```
LD    V200.7
AW=   VW208, 100
MOVB  2, VB301
MOVD  &VB101, VD302
MOVB  2, VB306
MOVW  0, VW307
NETW  VB300, 0
```

Subroutine Call (STL)

Format:

```
CALL n
```

Operands:
n: CPU 212: 0-15
CPU 214: 0-63

Description of operation:
The Subroutine Call (CALL) instruction transfers control to the subroutine (n).

Example:

```
LD    SM0.1
CALL  10
```

Conditional Return from Subroutine (STL)

Format:

```
CRET
```

Operands:
(none)

Description of operation:
The Conditional Return from Subroutine (CRET) instruction may be used to terminate a subroutine, based on the condition of the preceding logic.

Example:

```
LD    M14.3
CRET
```

Conditional End (STL)

Format:

```
END
```

Operands:
(none)

Description of operation:
The Conditional End (END) instruction terminates the main user program based on the condition of the preceding logic.

Example:

```
LD    SM5.0
STOP
END
```

For (STL)

Format:

```
FOR  INDEX,
INITIAL, FINAL
```

Operands:

INDEX (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC
INITIAL (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
FINAL (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:
The FOR instruction executes the code between the FOR and the NEXT. You must specify the current loop count (INDEX), the starting value (INITIAL), and the ending value (FINAL). If the starting value is greater than the final value, the loop is not executed. After each execution of the instructions between the FOR and the NEXT instruction, the INDEX value is incremented and the result is compared to the final value. If the INDEX is greater than the final value, the loop is terminated.

For example, given an INITIAL value of 1, and a FINAL value of 10, the instructions between the FOR and the NEXT are executed 10 times with the INDEX value incrementing 1,2,3, . . 10.

Example:

```
LD    I2.1
FOR   VW225, 1, 2
.
.
```

Jump to Label (STL)

Format:

```
JMP   n
```

Operands:
n: CPU 212: 0-63
CPU 214: 0-255

Description of operation:
The Jump to Label (JMP) instruction performs a branch to the specified label within the program.

Example:

```
LDN   SM0.2
JMP   4
.
.
.
LBL   4
```

Label (STL)

Format:

LBL *n*

Operands:

n: CPU 212: 0-63
CPU 214: 0-255

Description of operation:

The Label (LBL) instruction marks the location of the jump destination (n). The CPU 212 allows 64 labels, and the CPU 214 allows 256.

Example:

```
LDN   SM0.2
JMP   4
.
.
.
LBL   4
```

Main Program End (STL)

Format:

MEND

Operands:

(none)

Description of operation:

The Main Program End (MEND) instruction must be used to terminate the main user program.

Example:

```
.
.
MEND

SBR   10

LD    M14.3
CRET
.
.
```

Next (STL)

Format:

NEXT

Operands:

(none)

Description of operation:

The NEXT instruction marks the end of the FOR loop, and sets the top of stack to 1.

Example:

```
LD    I2.1
FOR   VW225, 1, 2
.
NEXT
```

No Operation (STL)

Format:

NOP *N*

Operands:

N: 0-255

Description of operation:

The No Operation (NOP) instruction has no effect on the user program execution. The operand *N* is a number from 0 - 255.

Example:

```
LDN   SM0.2
JMP   4

NOP

LBL   4
```

Unconditional Return from Subroutine (STL)

Format:

RET

Operands:

(none)

Description of operation:

The Unconditional Return from Subroutine (RET) instruction must be used to terminate each subroutine.

Example:

```
SBR   10

LD    M14.3
CRET

RET
```

Subroutine (STL)

Format:

SBR *n*

Operands:

n: CPU 212: 0-15
CPU 214: 0-63

Description of operation:

The Subroutine (SBR) instruction marks the beginning of the subroutine (n). The CPU 212 supports 16 subroutines, and the CPU 214 supports 64.

Example:

```
.
MEND

SBR   10

LD    M14.3
CRET
.
.
```

Stop (STL)

Format:

```
STOP
```

Operands:
(none)
Description of operation:
The Stop (STOP) instruction terminates execution of the user program by causing a transition to the Stop mode.
Example:

```
LD    SM5.0
STOP
```

Watchdog Reset (STL)

Format:

```
WDR
```

Operands:
(none)
Description of operation:
The Watchdog Reset (WDR) instruction allows the watchdog timer to be retriggered. This extends the time the scan is allowed to take without getting a watchdog error.
Example:

```
LD    M5.6
WDR
```

Rotate Left Double Word (STL)

Format:

```
RLD   IN, N
```

Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:
The Rotate Left Double Word (RLD) instruction rotates the double word value (IN) left by the shift count (N), and loads the result in IN.
SM1.0 (zero) = 1 if IN = 0
SM1.1 (overflow) = 1 if last bit rotated = 1
Example:

```
LD    I4.0
RLD   AC0, 2
SLW   VW200, 3
```

Rotate Left Word (STL)

Format:

```
RLW   IN, N
```

Operands:

IN (word):	VW, T,
N (byte):	VB, IB,

Description of operation:
The Rotate Left Word (RLW) instruction rotates the word value (IN) left by the shift count (N), and loads the result in IN.
SM1.0 (zero) = 1 if OUT = 0
SM1.1 (overflow) = 1 if last bit rotated = 1
Example:

```
LD    I4.0
RLD   AC0, 2
RLW   VW200, 3
```

Rotate Right Double Word (STL)

Format:

```
RRD  IN, N
```

Operands:

IN (Dword):	VD, ID, QD, MD, SMD, AC, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The Rotate Right Double Word (RRD) instruction rotates the double word value (IN) right by the shift count (N), and loads the result in IN.

SM1.0 (zero) = 1 if IN = 0
SM1.1 (overflow) = 1 if last bit rotated = 1

Example:

```
LD   I4.0
RRD  AC0, 2
SLW  VW200, 3
```

Rotate Right Word (STL)

Format:

```
RRW  IN, N
```

Operands:

IN (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC
N (byte):	VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The Rotate Right Word (RRW) instruction rotates the word value (IN) right by the shift count (N), and loads the result in IN.

SM1.0 (zero) = 1 if OUT = 0
SM1.1 (overflow) = 1 if last bit rotated = 1

Example:

```
LD   I4.0
RRW  AC0, 2
SLW  VW200, 3
```

Shift Register Bit (STL)

Format:

```
SHRB DATA, S_BIT, N
```

Operands:

DATA, S_BIT (bit):	I, Q, M, SM, T, C, V
N (byte):	VB, IB, QB, MB, SMB,Constant, *VD, *

Description of operation:

The Shift Register Bit (SHRB) instruction shifts the value of DATA into the shift register. S_BIT specifies the least-significant bit of the shift register. N specifies the length of the shift register and the direction of the shift (shift plus = N, shift minus = –N).

Shift Left Double Word (STL)

Format:

```
SLD  IN, N
```

Operands:

IN (Dword): VD, ID, QD, MD, SMD, AC, *VD, *AC

N (byte): VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The Shift Left Double Word (SLD) instruction shifts the double word value (IN) left by the shift count (N), and loads the result in IN.

SM1.0 (zero) = 1 if IN = 0

SM1.1 (overflow) = 1 if last bit shifted out = 1

Example:

```
LD   I4.0
SLD  AC0, 2
SLW  VW200, 3
```

Shift Left Word (STL)

Format:

```
SLW  IN, N
```

Operands:

IN (word): VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

N (byte): VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The Shift Left Word (SLW) instruction shifts the word value (IN) left by the shift count (N), and loads the result in IN.

SM1.0 (zero) = 1 if OUT = 0

SM1.1 (overflow) = 1 if last bit shifted out = 1

Shift Right Double Word (STL)

Format:

```
SRD  IN, N
```

Operands:

IN (Dword): VD, ID, QD, MD, SMD, AC, *VD, *AC

N (byte): VB, IB, QB, MB, SMB, AC, Constant, *VD, *AC

Description of operation:

The Shift Right Double Word (SRD) instruction shifts the double word value (IN) right by the shift count (N), and loads the result in IN.

SM1.0 (zero) = 1 if IN = 0

SM1.1 (overflow) = 1 if last bit shifted out = 1

Example:

```
LD   I4.0
SRD  AC0, 2
SLW  VW200, 3
```

Shift Right Word (STL)

Format:

```
SRW  IN, N
```

Operands:

IN (word):	VW, T, C, IW, QW, MW,*VD, *AC
N (byte):	VB, IB, QB, MB, SMB,

Description of operation:
The Shift Right Word (SRW) instruction shifts the word value (IN) right by the shift count (N), and loads the result in IN.

SM1.0 (zero) = 1 if OUT = 0
SM1.1 (overflow) = 1 if last bit shifted out = 1

Add To Table (STL)

Note: Table and Find instructions are supported by the CPU 214 only.

Format:

```
ATT  DATA, TABLE
```

Operands:

DATA (word):	VW, T
TABLE (word):	VW, T

Description of operation:
The Add To Table (ATT) instruction adds word values (DATA) to the table (TABLE). The first value of the table is the maximum table length (TL). The second value is the entry count (EC) that specifies the number of entries in the table. New data are added to the table after the last entry. Each time new data are added to the table, the entry count (EC) is incremented. If you try to overfill the table, the Table Full memory bit (SM1.4) is set.

Example:

```
LD   I3.0
ATT  VW100, VW200
```

First In First Out (STL)

Note: Table and Find instructions are supported by the CPU 214 only.

Format:

```
FIFO TABLE, DATA
```

Operands:

TABLE (word):	VW, T, C, IW, QW, MW, SMW, *VD, *AC
DATA (word):	VW, T, C, IW, QW, MW, SMW, AC, AQW, *VD, *AC

Description of operation:
The First In First Out (FIFO) instruction removes the first entry in the table (TABLE), and outputs the value to the location DATA. All other entries of the table are shifted up one location. The entry count (EC) in the table is decremented for each instruction execution. If you try to remove an entry from an empty table, the Table Empty memory bit (SM1.5) is set.

Example:

```
LD   I3.0
FIFO VW200, VW300
```

Find Less Than (STL)

Note: Table and Find instructions are supported by the CPU 214 only.

Format:

```
FND< SRC, PATRN,
INDX
```

Operands:

SRC (word):	VW, T, C, IW, QW, MW, SMW, *VD, *AC
PATRN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
INDX (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

The Find Less Than (FND<) instruction searches the table (SRC), starting with the table entry specified by INDX, for the data value (PATRN) that matches the find criteria.

If a match is found, the INDX points to the matching entry in the table. If a match is not found, the INDX has a value equal to the entry count. To find the next matching entry, the INDX must be incremented before the Find instruction is invoked again.

Example:

```
LD    I3.0
FND< VW202, 16#3130, AC1
```

Find Not Equal To (STL)

Note: Table and Find instructions are supported by the CPU 214 only.

Format:

```
FND<>      SRC,
PATRN,  INDX
```

Operands:

SRC (word):	VW, T, C, IW, QW, MW, SMW, *VD, *AC
PATRN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
INDX (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

The Find Not Equal To (FND<>) instruction searches the table (SRC), starting with the table entry specified by INDX, for the data value (PATRN) that matches the find criteria.

If a match is found, the INDX points to the matching entry in the table. If a match is not found, the INDX has a value equal to the entry count. To find the next matching entry, the INDX must be incremented before the Find instruction is invoked again.

Example:

```
LD     I3.0
FND<>       VW202, 16#3130, AC1
```

Find Equal To (STL)

Note: Table and Find instructions are supported by the CPU 214 only.

Format:

```
FND= SRC, PATRN,
INDX
```

Operands:

SRC (word):	VW, T, C, IW, QW, MW, SMW, *VD, *AC
PATRN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
INDX (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

The Find Equal To (FND=) instruction searches the table (SRC), starting with the table entry specified by INDX, for the data value (PATRN) that matches the find criteria.

If a match is found, the INDX points to the matching entry in the table. If a match is not found, the INDX has a value equal to the entry count. To find the next matching entry, the INDX must be incremented before the Find instruction is invoked again.

Example:

```
LD    I3.0
FND= VW202, 16#3130, AC1
```

Find Greater Than (STL)

Note: Table and Find instructions are supported by the CPU 214 only.

Format:

```
FND> SRC, PATRN,
INDX
```

Operands:

SRC (word):	VW, T, C, IW, QW, MW, SMW, *VD, *AC
PATRN (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
INDX (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of operation:

The Find Greater Than (FND>) instruction searches the table (SRC), starting with the table entry specified by INDX, for the data value (PATRN) that matches the find criteria.

If a match is found, the INDX points to the matching entry in the table. If a match is not found, the INDX has a value equal to the entry count. To find the next matching entry, the INDX must be incremented before the Find instruction is invoked again.

Last In First Out (STL)

Note: Table and Find instructions are supported by the CPU 214 only.

Format:

```
LIFO TABLE, DATA
```

Operands:

TABLE (word):	VW, T, C, IW, QW, MW, SMW, *VD, *AC
DATA (word):	VW, T, C, IW, QW, MW, SMW, AC, AQW, *VD, *AC

Description of operation:
The Last In First Out (LIFO) instruction removes the last entry in the table (TABLE), and outputs the value to the location DATA. The entry count (EC) in the table is decremented for each instruction execution. If you try to remove an entry from an empty table, the Table Empty memory bit (SM1.5) is set.

Example:

```
LD   I3.0
LIFO VW200, VW300
```

On Delay Timer (STL)

Format:

```
TON  Txxx, PT
```

Operands:

Txxx (word):	CPU 212: 32-63 CPU 214: 32-63, 96-127
PT (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:
The On-Delay Timer (TON) times up to the maximum value when the top of stack =1. When the current value (Txxx) is >= the Preset Time (PT), the timer bit (Txxx) turns on. It resets when the top of stack =0. Timing stops upon reaching the maximum value.

	CPU 212/214	CPU 214
1 ms	T32	T96
10 ms	T33-T36	T97-T100
100 ms	T37-T63	T101-T127

Example:

```
LD   I2.0
TON  T33, 3
```

Retentive On Delay Timer (STL)

Format:

```
TONR Txxx, PT
```

Operands:

Txxx (word):	CPU 212: 0-31 CPU 214: 0-31, 64-95
PT (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

Description of operation:
The Retentive On Delay Timer (TONR) times up to the maximum value when the top of stack =1. When the current value (Txxx) is >= the Preset Time (PT), the timer bit (Txxx) turns on. Timing stops when the top of stack =0, or upon reaching the maximum value.

	CPU 212/214	CPU 214
1 ms	T0	T64
10 ms	T1-T4	T65-T68
100 ms	T5-T31	T69-T95

Example:

```
LD    I2.1
TONR  T2, 1
```

AND Word (STL)

Format:

```
ANDW IN1, IN2
```

Operands:

IN1 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
IN2 (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of STL operation:
The AND Word (ANDW) instruction logically ANDs the corresponding bits of two words IN1, IN2, and loads the result in the word IN2.

Example:

```
LD    I4.0
ANDW  AC1, AC0
```

OR Word (STL)

Format:

```
ORW  IN1, IN2
```

Operands:

IN1 (word):	VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC
IN2 (word):	VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of STL operation:
The OR Word (ORW) instruction logically ORs the corresponding bits of two words IN1, IN2, and loads the result in the word IN2.

Example:

```
LD    I4.0
ORW   AC1, VW100
```

Exclusive OR Word (STL)

Format:

XORW *IN1, IN2*

Operands:

IN1 (word): VW, T, C, IW, QW, MW, SMW, AC, AIW, Constant, *VD, *AC

IN2 (word): VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of STL operation:

The Exclusive OR Word (XORW) instruction logically XORs the corresponding bits of two words IN1, IN2, and loads the result in the word IN2.

Example:

```
LD    I4.0
XORW AC1, VW100
```

AND Double Word (STL)

Format:

ANDD *IN1, IN2*

Operands:

IN1 (Dword): VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC

IN2 (Dword): VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of STL operation:

The AND Dword (ANDD) instruction logically ANDs the corresponding bits of two double words IN1, IN2, and loads the result in the double word IN2.

Example:

```
LD    I4.0
ANDD AC1, AC0
```

OR Double Word (STL)

Format:

ORD *IN1, IN2*

Operands:

IN1 (Dword): VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC

IN2 (Dword): VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of STL operation:

The OR Dword (ORD) instruction logically ORs the corresponding bits of two double words IN1, IN2, and loads the result in the double word IN2.

Example:

```
LD    I4.0
ORD  AC1, VD100
```

Exclusive OR Double Word (STL)

Format:

XORD *IN1, IN2*

Operands:

IN1 (Dword): VD, ID, QD, MD, SMD, AC, HC, Constant, *VD, *AC

IN2 (Dword): VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of STL operation:

The Exclusive OR Dword (XORD) instruction logically XORs the corresponding bits of two double words IN1, IN2, and loads the result in the double word IN2.

Example:

```
LD    I4.0
XORD AC1, VD100
```

Increment Word (STL)

Format:

INCW *IN*

Operands:

IN (word): VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of STL operation:

The Increment Word (INCW) instruction adds 1 to the input word value IN, and loads the result in that word.

IN + 1 = IN

Example:

```
LD    I4.0
INCW AC0
```

Decrement Word (STL)

Format:

DECW *IN*

Operands:

IN (word): VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of STL operation:

The Decrement Word (DECW) instruction subtracts 1 from the input word value IN, and loads the result in that word.

IN - 1 = IN

Example:

```
LD    I4.0
DECW VW100
```

Increment Double Word (STL)

Format:

INCD *IN*

Operands:

IN (Dword): VD, ID

Description of STL operation:

The Increment Dword (INCD) instruction adds 1 to the input double word value IN, and loads the result in that double word.

IN + 1 = IN

Example:

```
LD    I4.0
INCD AC0
```

Decrement Double Word (STL)

Format:

DECD *IN*

Operands:

IN (Dword): VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of STL operation:

The Decrement Dword (DECD) instruction subtracts 1 from the input double word value IN, and loads the result in that double word.

IN - 1 = IN

Example:

```
LD    I4.0
DECD  VD100
```

Invert Word (STL)

Format:

INVW *IN*

Operands:

IN (word): VW, T, C, IW, QW, MW, SMW, AC, *VD, *AC

Description of STL operation:

The Invert Word (INVW) instruction takes the Ones Complement of the input word value IN, and loads the result in that word.

Example:

```
LD    I4.0
INVW  AC0
```

Invert Double Word (STL)

Format:

INVD *IN*

Operands:

IN (Dword): VD, ID, QD, MD, SMD, AC, *VD, *AC

Description of STL operation:

The Invert Dword (INVD) instruction takes the ones complement of the input double word value IN, and loads the result in that double word.

Example:

```
LD    I4.0
INVD  AC0
```

Reference

1. Programmable Logic Controllers and their Engineering Applications
   ALAN J. CRISPIN

2. Programmable Controllers Operation and Application
   IAN G. WARNOCK

3. Programmable Controllers an Engineer's Guide
   E. A. PARR

4. Lecture Notes
   ÖZGÜR ÖZERDEM