BULLETIN OF THE POLISH ACADEMY OF SCIENCES
TECHNICAL SCIENCES, Vol. 70(2), 2022, Article number: e140554
DOI: 10.24425/bpasts.2022.140554

Compiler and virtual machine of a multiplatform control environment

Jan SADOLEWSKI and Bartosz TRYBUS*

Department of Computer and Control Engineering, Rzeszow University of Technology, ul. W. Pola 2, 35-959 Rzeszow, Poland

Abstract. Design and operation of a compiler and virtual machine, being the essential components of a multiplatform control programming environment, are presented. The compiler translates source programs written in Structured Text language of the IEC 61131-3 standard into executable code in a dedicated intermediate language. The virtual machine, i.e. a specially designed processor implemented in software, is a runtime part of the environment executing the code in real time. Due to memory-to-memory operation principle the machine is able to process various data types defined in the standard. The focus is given on overloading and extensibility of the functions, as well as on uniform invocations of Program Organization Units. By selection of addressing mode, the environment can be deployed on multiple hardware platforms, beginning from 8-bit microcontrollers up to 32/64-bit industrial PCs. Industrial applications are indicated.

Key words: control environment; IEC 61131-3 standard; intermediate language; compiler; runtime virtual machine.

1. INTRODUCTION

The IEC 61131-3 standard [1] on programming languages for control systems defines five languages, namely textual IL, ST, graphical LD, FBD, and mixed SFC. Software environment implementing the standard in a particular controller consists of two essential components, i.e. compiler and runtime. The compiler translates a source program written in one of the IEC languages (61131-3 dropped for brevity) into an executable code transferred to the runtime in the controller processor. The runtime executes the code in real time with a given cycle.

Three approaches to implementation of the standard are currently in use. The first one directly compiles the IEC programs to processor code. Relatively uncomplicated runtime executes the code quickly, so the approach is used mostly by established manufacturers. However, it is a single processor solution since change of CPU requires a new compiler.

The second approach involves compilation of IEC programs to C/C++ and then another compiler translates it to the target processor. Beremiz, a solution of academic origin, provides capabilities to compile IEC projects to several hardware platforms [2, 3]. GEB Automation [4] emphasizes educational applications of its software by using open-source C/C++ compilers in the second step. Despite that the approach offers multiplatform applications, the industry often remains reluctant to the two-step toolchain and open-source compilers.

The third approach is based on virtual machine (VM) concept, i.e. an emulated abstract processor implemented in software that executes certain intermediate code to which source programs are compiled. The VM concept was originally introduced in ISaGRAF [5] with an intermediate Target Independent Code. STRATON [6] also applies this approach although without disclosing details. Multiplatform applications, independent from the target CPU, are benefits of the approach while less time efficiency due to operation of the VM itself is a disadvantage.

For research reasons, the VM-based approach has been especially interesting for academic community, initially with the assembler-like IL as the intermediate language. In particular, compiler design involving translation of the four other IEC languages to IL is described in [7]. Since IL instructions store results in virtual registers, the authors stress importance of register allocation by the compiler. A RISC processor executes the final code. A VM of [8] continuously decodes IL instructions and, if semantically correct, executes one after another. The machine is written in C and runs on 8-bit CPU. An intermediate assembler-like language not tied up to IL is supported by a VM of [9]. The VM has running stack and some x86-type registers. It is written in C and runs on embedded ARM as a SoftPLC.

Other solutions, close in fact to the second approach, apply common intermediate languages such as Java or C#, followed by ready-to-use Java Virtual Machine (JVM) or .NET Common Language Runtime (CLR). Multiple target platforms are thus enabled. In [10], ST and FBD programs are compiled to Java bytecode transferred to embedded devices running the JVM. Likewise in [11], one module of the environment compiles IEC projects into C# and another deploys them to CLR VMs in Windows or Linux. Java, C/C++ or Objective-C may be selected as the intermediate language in a VM implementation for secure Internet of Things [12].

In addition to VMs implemented in software, a dedicated runtime can also be designed in hardware, as demonstrated in [13] for a Toshiba language processor. Nowadays hardware processors are typically designed in FPGA technology. For ex-

*e-mail: btrybus@prz.edu.pl

Manuscript submitted 2021-07-30, revised 2022-01-08, initially accepted for publication 2022-01-25, published in April 2022.

ample, a PLC design as a System-on-a-Chip with IL as the native language is described in [14].

One may also add that a general purpose LLVM (initially Low-Level Virtual Machine) infrastructure provides a toolchain framework to develop efficient compilers [15]. The LLVM is an intermediate stage between the source code and the native code of a target processor. To apply LLVM the source code, originally in C or C++, must be converted into an intermediate representation (IR), which is a kind of a high-level assembler. Then the LLVM compiles and optimizes such front-end to generate intermediate code in IR. Finally, the IR code is converted into native code by a code generator (back-end) dedicated to a particular processor. Adaptation of the Beremiz front-end [2, 3] to create LLVM compilers of IEC languages has been described in [16] recently.

Working with a few colleagues, the authors of this paper began developing a control programming environment named CPDev (Control Program Developer) over 10 years ago, assuming initially that it would not be restricted to any particular processor or hardware solution. This excluded the first of the three approaches indicated above. The second approach, although appropriate for teaching and research, suffers in the case of industrial applications. Therefore, the third approach involving a custom-designed VM remained an appropriate choice.

In the proposed solution the intermediate code generated by the compiler is portable, so there is no need to generate the native code or LLVM back-end separately for each type of the target processor or architecture. This is advantageous in distributed systems involving controllers with various processors where portable intermediate code can be generated and deployed by the same tool. Since the layout of the code and data in memory is identical for all targets, debugging is unified as code interpretation is the same in the simulator, debugger and the target controller. The disadvantages compared to native compilers or LLVM-based solutions are the significantly longer execution time of the programs interpreted by the virtual machine and necessity of development of custom compilation, optimization and debugging tools.

To enable complex programming, the textual ST was selected as the first language supported by the CPDev environment. Graphical diagrams in LD, FBD, and SFC are translated to their textual representation in ST and then compiled. IL programs are executed after converting to the portable code. The original and last versions of CPDev are described in [17] and [18], respectively. Petri-net models of the VM and a list of prototype functions to handle general target hardware can be found in [19]. The VM source code for some popular platforms is available in the public repository at [20].

The available literature on environments and tools for IEC programming and runtime, including [17–19], reports on general characteristics, functionalities, and implementations rather than how the environments have been created. Therefore, the novelty of this paper rests in the presentation of solutions applied in CPDev compiler and virtual machine to such issues as overloading and extensibility of IEC functions, components of semantic validation of the parsed code, nested invocations of POUs (Program Organization Units), selection of addressing mode for multiplatform implementations, and some universal C code.

The paper is organized as follows. The next section presents a concept of the intermediate language called Virtual Machine Assembler (VMASM). Section 3 describes the scanner and parser with semantic validation, followed by an explanation of nested POU invocations. Code generation involving digital identifiers of the VMASM instructions is presented in Section 4. The identifiers express overloading and extensibility. VM architecture, operation algorithm, and exemplary C code including automatic implementation of 16- or 32-bit addressing are described in Section 5. Two industrial implementations are reviewed in Section 6. The last section summarizes the paper.

2. CONCEPT OF THE INTERMEDIATE LANGUAGE

The choice of how the virtual machine is going to operate determines the VMASM language. A memory-to-memory operation principle is chosen due to numerous data types. This principle and the extensibility of some IEC functions determine the syntax of the VMASM instructions. Assuming control orientation of the VM, one part of the instructions consists of direct counterparts of the IEC standard functions, whereas the other part involves assembler-like jumps, memory copying, subprogram calls, etc., called system procedures.

2.1. Expression tree

Various sizes and types of IEC data imply that it would be difficult to design a conventional accumulator- or register-based architecture of the VM. The difficulty may be overcome by choosing an architecture based on memory-to-memory operation principle, which means that the result of instruction execution is directly uploaded into the memory. Such a solution avoids the register scarcity problem already encountered in [7].

As an introductory example, consider control of a `MOTOR` turned on after pressing `START`, and turned off after pressing `STOP` or when an `ALARM` appears. The corresponding ST assignment may have the form

```
MOTOR := (START OR MOTOR) AND NOT (STOP OR ALARM);
```

A tree representing the expression on the right side is shown in Fig 1.

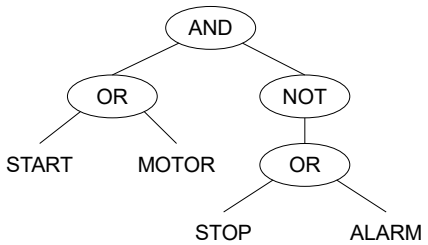

Fig. 1. Expression tree for the `MOTOR` example

Let the identifiers of ST variables and IEC functions `AND`, `OR`, `NOT`, etc. remain unchanged in the VMASM language. Notice that processing the tree will require some additional compiler-generated variables. They can be distinguished from the ST

ones by including a symbol not admitted in ST, for instance question mark ? placed at the beginning. So ?R1 may mean a result of the first OR in Fig. 1. If needed, another ? may appear somewhere inside such identifier.

Due to the memory-to-memory operation, the result ?R1 must appear on the operand list of the first OR, either at the end or at the beginning. However, since OR is an extensible function, placing the result at the end would make its memory location dependent on the number of operands, whereas the beginning provides a convenient constant location. Hence the instruction implementing the first OR may look as follows

```
OR ?R1, START, MOTOR
```

Recall that for example the x86 assembler also places instruction results at the beginning of operand list [21].

2.2. VMASM syntax

Besides the compiler-generated variables, labels for jump instructions are also needed. Therefore the syntax of the VMASM instructions has the form

```
[:label] instruction [operand0] [, operand1]
   [, operand2] ...
```

The label is optional, number of operands depends on instruction, although it is limited to 16 for practical reasons. In the case of a function like OR the operand0 means a result, while in the direct jump JMP it will be a label. The label itself, as a compiler-generated identifier, has the question mark ? right after the colon. Thus, :?L1 may be an example of the label.

Coming back to the MOTOR example, VMASM mnemonic code implementing the expression tree of Fig. 1 may look as in Fig. 2. JZ is the common Jump-if-Zero, whereas MCD (Memory-Copy-Data) copies the given number of data bytes (operand1) and the content (operand2) to the result (operand0). Notice that if the result ?R1 of the first OR is 0 then the following JZ jumps to :?L1 to set MOTOR to 0, thus abandoning five subsequent instructions. This is an example of the so-called lazy evaluation of an expression when a partial outcome indicates the final result [22]. The compiler presented here translates ST expressions for lazy evaluation.

```
        OR ?R1, START, MOTOR
        JZ ?R1, :?L1
        OR ?R2, STOP, ALARM
        NOT ?R3, ?R2
        JZ ?R3, :?L1
        MCD MOTOR, 1, 1
        JMP :?L2
:?L1    MCD MOTOR, 1, 0
:?L2    ...
```

Fig. 2. Translation of the MOTOR example

2.3. System procedures

As indicated before, the VMASM instructions consist of IEC function counterparts and assembler-like system procedures [17, 19]. The total of 45 procedures can be divided into the following groups:

- jumps (8),
- subprograms (6),
- initializations (5),
- copying (4),
- arrays and pointers (17),
- assertions (3),
- other (5).

The subprogram group includes CALB (CALL-program-Block) and RETURN instructions that handle invocations of POUs. Assertions are used for debugging.

Note that the intermediate language of [9] consists of 54 instructions with 0, 1 or 2 operands, including boolean, mathematical, bit and other operations. These are executed here by the IEC function counterparts.

3. COMPILER

Scanner and parser involving semantic validation transform an ST project into VMASM mnemonic code. User-defined functions, function blocks, and programs are invoked in a uniform way.

3.1. Scanner and parser

The compiler converts an XML source file with an ST project into an executable VMASM file in the hexadecimal format. Diagram of the compiler operation is shown in Fig. 3.

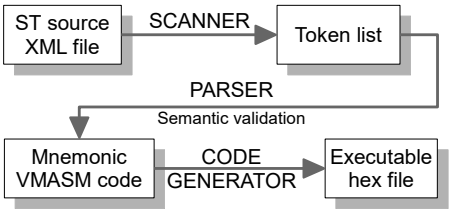

Fig. 3. Components of the ST-to-VMASM compiler

The scanner analyzes character stream from the source file and decomposes it into lexical tokens classified into categories such as identifiers, keywords, operators, constants, etc. The tokens with categories are collected on a list passed to the parser.

After checking whether the tokens contain valid characters only, the parser proceeds with a top-down syntax-directed translation [23] by grouping the tokens into constructs according to ST language grammar. If the constructs are correct the parser builds an internal abstract syntax tree.

In the next phase the tree is semantically validated with respect to variables, expressions, invocations, and nesting [3, 24] by performing the following checks:

- variables: declaration before usage, visibility in the POU context, consistency of data type and initial value,
- expressions: compatibility of operators and operand data types,
- invocations: declaration of POU in the project or library, number of operands and their types consistent with POU interfaces, user-defined function cannot invoke function blocks,
- nesting: nested IFs and invocations must follow the recursive order.

Having the validated tree the parser replaces the ST constructs with sets of mnemonic VMASM instructions like the ones in Fig. 2. To do so it employs the built-in elementary data types and the list of instructions. Some translations require introduction of additional variables and labels. An expression involving operators is first transformed into a subtree and then an appropriate routine converts it into VMASM code. Multi-element types such as arrays and structures, and POUs acquired from libraries are also parsed. Finally the code is consolidated with other mnemonic codes and written in a special text format.

3.2. Classes of the internal data

Basic elements of the compiler are designed as classes in the C# language. The scanner generates objects of the `BasicToken` class. The ST tokens acquired by the parser become objects of relevant classes such as `STVariable`, `STFunctions`, `STLibrary`, etc. The classes inherit from an abstract `STIdentifier` class.

The mnemonic instructions passed from the parser to the code generator are represented by instances of the `VMInstruction` class. The `VMOperand` list is a field in this class.

During compilation, object identifiers are collected on appropriate lists. So there is a list of global identifiers, lists of local identifiers for POUs, etc. By applying predictors to find identifiers on the lists the cumbersome hash tables are avoided.

3.3. POU invocations and tasks

Invocations of all POU types, i.e. user-defined functions, function blocks, and programs are executed by the same `CALB...RETURN` pairs. A single call has the form

```
CALB InstPtr, FunBlockAddr
```

where `InstPtr` denotes an instance and `FunBlockAddr` an address of the POU code, here indicating a function block. Definitions of `CALB`, `RETURN`, and two other instructions are in Table 1.

Table 1
Definitions of three VMASM procedures and OR function

```
<sysproc name="CALB" vmcode="1C16">
 <op no="0" name="inst" type=":rdlabel"/>
 <op no="1" name="clbl" type=":gclabel"/>
</sysproc>

<sysproc name="RETURN" vmcode="1C13">
</sysproc>

<sysproc name="TRML" vmcode="1C1D">
 <op no="0" name="AddressStart" type=":gclabel"/>
</sysproc>

<function name="OR" vmcode="09*0" return="BOOL">
 <op no="*" name="arg*" type="BOOL"/>
</function>
```

Although the VM architecture will be explained only later in Section 5, it is now indicated that the VM contains registers with code and data address pointers, called code and data registers for brevity. The type `:rdlabel` (relative-data-label) of `InstPtr` above means an address in the data memory relative to the content of the data register, whereas `:gcdlabel` (global-code-label) of `FunBlockAddr` is a direct address (global) in the code memory. Stack emulation mechanism triggered by `CALB` pushes the registers on corresponding stacks. `RETURN` pulls the stacks.

In the case of a user-defined function, the data memory address of the operating area for local variables must be the same for all invocations of the function. The address is jointly indicated by the relative `InstPtr` and the data register.

Each program executed by the VM must be able to access global variables. Let 0 (zero) be the beginning address of the area for the global variables and let the data register content while invoking a program be also 0. Hence the invocation of a program accessing the global variables may have the form

```
CALB 0, ProgramAddr
```

Since the invoked program ends up with `RETURN`, the next program will be also called with the zero data register.

For the program invocation as above, code of a task that cyclically executes some programs `PRG1`, `PRG2`, ... may have the form shown in Fig. 4.

```
    :?TSK1   CALB 0, :?PRG1?CODE
             CALB 0, :?PRG2?CODE
             ...
             TRML :?TSK1
```

Fig. 4. Code of the task loop

The `TRML` instruction defined in Table 1 terminates the task. Its `AddressStart` indicates the starting point used by the runtime when the task is resumed, i.e. the label `:?TSK1` is Fig. 4.

A task can be executed in one of three modes:

- cyclic – as a loop with constant cycle time,
- continuous – loop with next execution resumed immediately,
- triggered – execution when a condition is met.

A control project may involve a loop task and several single execution tasks triggered by some events.

Before resuming a task the `TRML` instruction activates execution of the VM internal code dependent on implementation. Typically it involves I/O handling, communications, and testing [19]. When the internal code is completed, `AddressStart` is fetched to the code register and VM begins execution of the task.

3.4. Example with function blocks

The `MOTOR` control of Section 2 will now be implemented by means of two nested function blocks according to the diagram of Fig. 5a. The `PRG` program invokes an instance `SS1` of a user-defined function block `SS` which nests the `OR` function and an instance `RS1` of the standard RS flip-flop. Implementation in the CPDev environment as a project `PRO` in which the `PRG` program

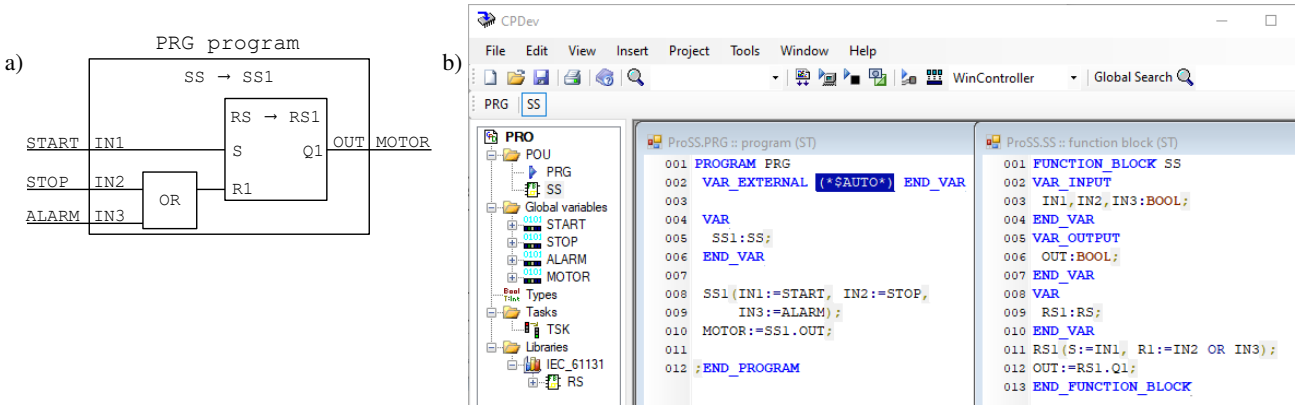

Fig. 5. a) Diagram of the example; b) implementation in the CPDev environment

is executed by the task TSK is shown in Fig. 5b. Global variables START, STOP, etc. are imported by the (*$AUTO*) directive. The IEC_61131 library includes the RS flip-flop.

Mnemonic instructions of the TSK loop and the ones implementing nesting by means of CALBs are shown in Table 2 (project name PRO begins the labels). The dropped code is characterized in comments with final RETURNs.

Table 2
Function block nesting in the mnemonic code

Entity	Mnemonics for nested POUs
TSK	:?PRO.TSK?TSKLOOP CALB #0000, :?PRO.PRG?CODE TRML :?PRO.TSK?TSKLOOP
	:?IEC_61131.RS?CODE ... /* RS code, RETURN */
RS1	:?PRO.SS?CODE ... /* S:=IN1, R1:=OR(IN2, IN3) */
SS1	CALB RS1, :?IEC_61131.RS?CODE ... /* OUT:=Q1, RETURN */
PRG	:?PRO.PRG?CODE ... /* IN1:=START, ... */ CALB SS1, :?PRO.SS?CODE ... /* MOTOR:=OUT, RETURN */

4. EXECUTABLE CODE

After scanning and parsing the code generator converts the consolidated mnemonic code into the executable hexadecimal file (Fig. 3). Digital identifiers of the VMASM functions take into account overloading and extensibility.

4.1. Code generator

The generator replaces instruction mnemonics by corresponding digital identifiers and addresses of operands or constants. To do so, the generator employs a Library Configuration File (LCF) with instruction definitions as the ones in Table 1, where the vmcodes denote digital identifiers.

A vmcode is a two-byte entity composed of the group ig and type it components shown in Fig. 6. In the case of a function, ig denotes the name of an overloaded group, such as OR, NOT, ADD, etc., whereas it specifies the number num of input operands and return type. By changing it the specific functions of the overloaded group are selected. An asterisk * as num indicates extensible number of inputs, as in the case of vmcode=09*0 for OR (09) that returns BOOL (0) (Table 1). As another example, NOT (05) for one (1) WORD operand (4) has vmcode=0514. System procedures such as JMP, CALB, RETURN, etc. belong to one group ig=1C and are also selected by it.

vmcode

ig	it	
group	num	type

Fig. 6. Digital identifier of a function

4.2. Memory allocation

Besides the instructions, the LCF file also specifies particular implementation by size of the address equal to two or four bytes (16- or 32-bit addressing). Since the vmcode consists of two bytes, so the code memory section occupied by a single instruction with n operands is either 2+n*2 or 2+n*4 bytes, respectively.

To make allocation of operating areas in the data memory to POUs easier to verify during debugging it is assumed that each area must be a multiple of a standard segment. The size of the segment equals the size of the longest elementary data which may be a single operand. So it is 8 bytes, as for LREAL or DATE_AND_TIME. The operating areas are determined at the final stage of parsing.

When a POU is invoked during execution the stack mechanism automatically increases the data register by the size of relevant operating area. Nested invocations enlarge this register accordingly.

4.3. Remarks on hexadecimal code

Because the POUs of the exemplary project use only a few BOOL variables, the operating areas of standard eight bytes

suffice for each of them. So when the PRG program invokes the block SS1, the data register is increased to 8, and when SS1 subsequently invokes the RS1 flip-flop it is farther increased to 16. Hence the global variables START to MOTOR of the project are assigned the addresses 0 to 3, the interface variables INT to OUT of SS1 the addresses 8 to 11, and S,R1,Q1 of RS1 the addresses 16, 17, 18.

Hexadecimal representations of two initial instructions from Table 2, for instance placed by the compiler at the addresses :001E and :0024 (hex), respectively, are as follows

```
CALB    :001E  1C16 0000 9200
TRML    :0024  1C1D 1E00
```

The lines begin with the vmcodes from Table 1. The subsequent addresses are in the little endian form, so 1E00 in TRML means :001E in the preceding CALB. Likewise 9200 in CALB means the address :0092 allocated to :?PRO.PRG?CODE (Table 2).

5. VIRTUAL MACHINE

Architecture of the virtual machine reflects the memory-to-memory operation. By means of the group and type identifiers selected from the vmcode, a proper instruction is acquired and processed. According to the length of addresses the VM can be deployed in 8, 16 or 32/64-bit platforms.

5.1. Architecture

The VM software-implemented architecture shown in Fig. 7 involves Harvard separate code and data memories, instruction processing module, code and data stacks, registers and pointers, and a target platform interface. The instruction processing module fetches the instructions from the code memory, executes them acquiring the operands from data or code memories and, in case of the functions, stores the results in the data memory. Assembler-like system procedures change internal state of the VM components. Stack mechanism composed of the two stacks

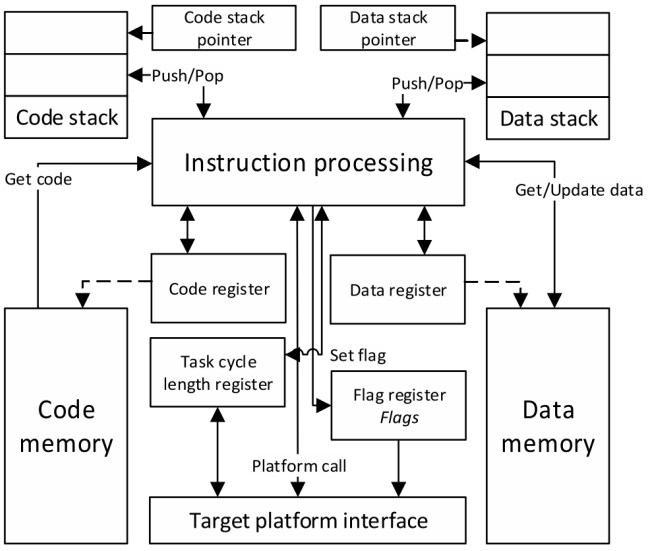

Fig. 7. Architecture of the virtual machine

and pointers supports invocations of POUs. The target platform interface involves low-level functions dependent on hardware platform and operating system [19].

The code register, in other words instruction pointer, contains addresses of instructions and operands. The data register (data pointer) keeps the base address of the POU operating area for relative addressing. The task cycle length register indicates the time till the end of the cycle. The flags register contains status flags for operating modes, errors, and unusual events.

The components can be implemented in 16- or 32-bit versions, depending on the maximum size of code and data memories (64 kB vs. 4 GB). So a VM for a given hardware is specified by the size of address in the LCF configuration file and the target platform interface.

5.2. Instruction processing

Each line of the executable code acquired by the instruction processing module begins with vmcode composed of the group and type identifiers ig and it (bytes). An algorithm for processing a single line is shown in Fig. 8. Assuming that the code register initially points out to the vmcode, the algorithm begins with fetching ig and it followed by incrementations of the code register. After that the register points to the first operand of the instruction.

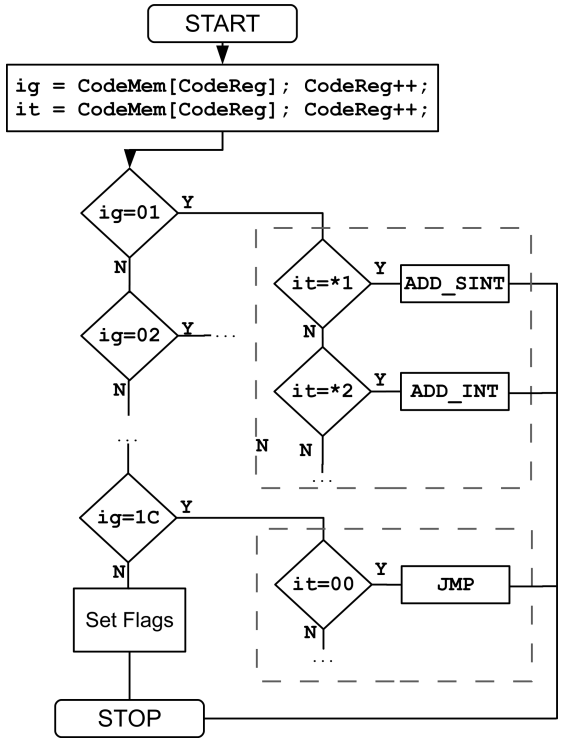

Fig. 8. Algorithm for processing single line of code

The following part matches the group identifier ig to particular group of functions or to procedures. In the case of a function the algorithm matches it to relevant type and executes the code. The procedures and type conversions are selected by it. Implementation of the algorithm in C is fairly straightforward by means of the switch() statements.

5.3. Addresses and instructions in C

Let `unsigned short ADDRESS` or `unsigned long ADDRESS` define 16- or 32-bit implementation of the VM according to the LCF file. Suppose the code register points out to an operand of instruction, initially to the first one. `GetCodeAddress` function shown in Fig. 9 transforms the standard byte pointer `*CodeMemory[CodeReg]` into `*ADDRESS` pointer and copies the content into a 16- or 32-bit variable `addr` being returned. The code register is incremented accordingly to point out to the next operand.

```
ADDRESS GetCodeAddress(void)
{
  ADDRESS addr =
   (*(ADDRESS*)(CodeMemory[CodeReg]));
  CodeReg += sizeof(ADDRESS);
  return addr;
}
```

Fig. 9. Getting the operand address

If the operand means a label or an immediate value then its address in the code memory is acquired by

```
ADDRESS operand = GetCodeAddress();
```

However, if the operand is a variable or a POU instance then the sum

```
ADDRESS operand = DataReg + GetCodeAddress();
```

involving the data register gives a global address in the data memory.

An overloaded function may be implemented by calling a single C function for all relevant data types to avoid repetitions of rather similar code. In turn that single function may call a code macrodefinition parametrized with respect to type to execute calculations.

An example of `IG_ADD_01` function that implements the overloaded extensible `ADD` is shown in Fig. 10. The function

```
void IG_ADD_01(BYTE it)
{
  switch (it & 0x0F)
  {
    ADD_TYPE(SINT);
    ADD_TYPE(INT);
    ADD_TYPE(DINT);
    ADD_TYPE(LINT);
    ...   /* other types */
  }
  return;
}

#define ADD_TYPE(TYPE) \
case IT_ADD_##TYPE & 0x000F: \
{ \
 BYTE num = it >> 4; \
 ...
} \
break;
```

Fig. 10. Outline of the ADD implementation

recognizes particular type by masking `it&0x0F` (see Fig. 6) and switches to execution by `ADD_TYPE`. Here the shift `it>>4` acquires the number `num` of inputs and implements the adding in a loop (not shown). Note that the `ADD_TYPE` definition retains `case` and `break` for the `switch` from `IG_ADD_01`. `0x000F` in `case` indicates that up to 16 IEC data types can be processed by `ADD`.

All system procedures from the group `ig=1C` are handled by the function `IG_SYSPROC_1C` outlined in Fig. 11, with self-explanatory `CALB` shown only.

```
void IG_SYSPROC_1C(BYTE it)
{
 switch(it)
 {
  ...
  case 0x16: /* CALB call a function block */
  {
   ADDRESS instaddr =
   dataReg + GetCodeAddress();
   ADDRESS clbl = GetCodeAddress();
   push_CodeStack(codeReg);
   push_DataStack(dataReg);
   dataReg = instaddr;
   codeReg = clbl;
  }
  break;
  ...
 }
 return;
```

Fig. 11. System procedure group with CALB

6. IMPLEMENTATIONS

The original CPDev environment was implemented for the first time by Lumel [25] in a small distributed control system described in [17, 19]. The system involved 8-bit AVR processor and several remote I/O modules with serial Modbus communication. The system is still installed by ISS [26], Philippines.

Ship navigation and automation systems from Praxis [27], the Netherlands, are the most significant implementations of the environment, providing for over decade continuing motivation for further development. Propulsion control (Fig. 12a, version for yachts and small ships), power management, heading control (autopilot), and a few other systems are connected to the ship main computer by redundant Ethernet. Each system consists of a control processor, I/O units, and a TFT touch panel, each of them equipped with 32-bit ARMs. CAN is applied for communication with I/O units and Ethernet (redundant) for TFT panels. TFT displays are integrated with control software by means of global variables. NMEA serial/Ethernet protocol is used by marine electronic devices. Programs are written in ST language.

Remote telecontrol units (Fig. 12b) from iGrid [28], Spain, are applied for substation automation, medium voltage control, and grid reconfiguration by means of triggered-mode tasks. A unit involves an ARM processor, I/O board, and a few communication interfaces including IEC 60870 and 61850 dedicated for power systems. LD and FBD languages are preferred.

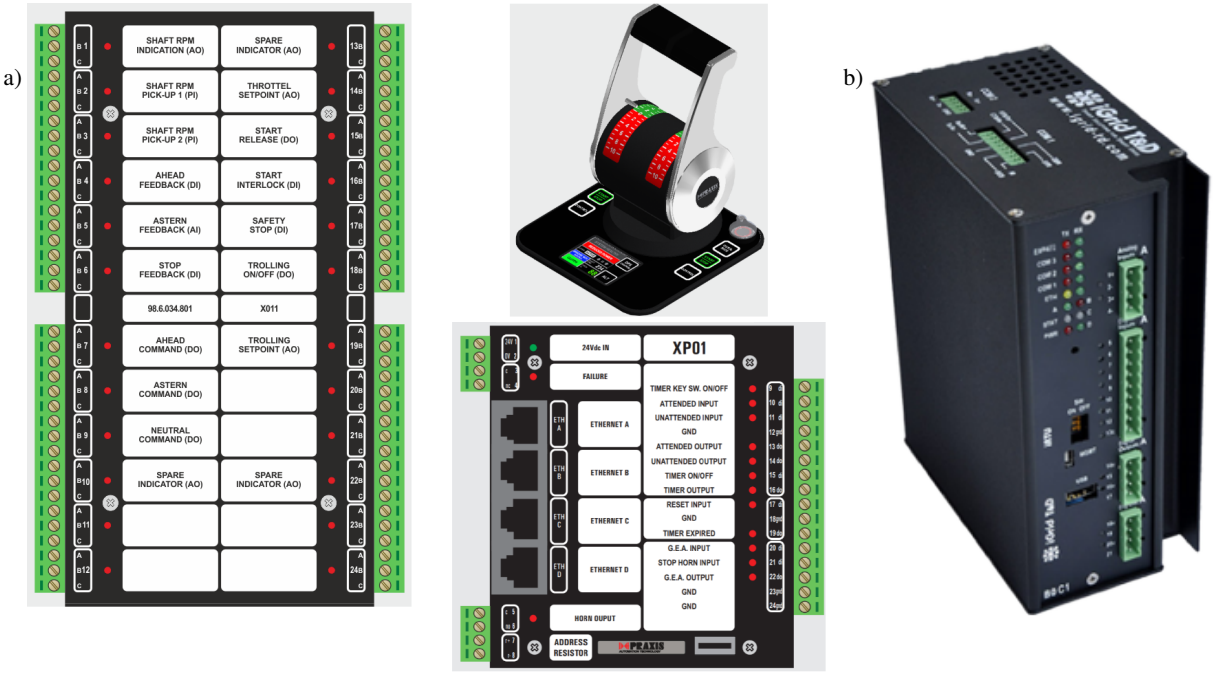

Fig. 12. a) Propulsion Control System [27]: control processor with I/O (left) and communication terminals (bottom), lever with TFT 2.5" operator panel; b) Remote Telecontrol Unit [28]

Some information on runtime for complex software involving several cooperating VMs running with different cycles and on domestic implementations can be found in [18]. The runtime is called WinController and runs on 32/64-bit industrial PC with Windows Embedded OS.

7. CONCLUSION

The paper has presented solutions applied in the compiler and virtual machine of the CPDev control environment to a few issues related to the IEC standard, namely overloading and extensibility, semantic verification, nested invocations, selection of addressing mode, and outline of some code. Standard IEC functions are built into the VMASM language.

The platform-independent code generated by the compiler facilitates deployment on a variety of hardware, from 8-bit microcontrollers to 32/64-bit industrial PCs.

ACKNOWLEDGEMENTS

Contribution of D. Rzońca, A. Stec, Z. Świder and L. Trybus to development of the CPDev environment is acknowledged.

This project is financed by the Minister of Education and Science of the Republic of Poland within the "Regional Initiative of Excellence" program for years 2019–2022. Project number 027/RID/2018/19, amount granted 11 999 900 PLN.

REFERENCES

[1] International Electrotechnical Commission, "EN 611313:2013 – Programmable controllers – Part 3: Programming languages," *European Committee for Electrotechnical Standardization, Tech. Rep*, 2013.

[2] *Beremiz integrated development environment*. [Online]. Available: www.beremiz.org (Accessed 2021-11-24).

[3] E. Tisserant, L. Bessard, and M. de Sousa, "An Open Source IEC 61131-3 Integrated Development Environment," in *IEEE Int. Conf. on Ind. Inform.*, 2007, pp. 183–187, doi: 10.1109/ INDIN.2007.4384753.

[4] GEB Automation, *GEB Automation IDE Guide*. [Online]. Available: www.gebautomation.org (Accessed 2021-11-24).

[5] Rockwell Automation, *ISaGRAF Workbench*. [Online]. Available: www.isagraf.com (Accessed 2021-11-24).

[6] COPA-DATA France, *STRATON*. [Online]. Available: www.straton-plc.com (Accessed 2021-11-24).

[7] H.S. Kim, J.Y. Lee, and W.H. Kwon, "A compiler design for IEC 1131-3 standard languages of programmable logic controllers," in *SICE'99 Ann. Conf.*, 1999, pp. 1155–1160, doi: 10.1109/SICE.1999.788715.

[8] C. Zhou and H. Chen, "Development of a PLC Virtual Machine Orienting IEC 61131-3 Standard," in *Int. Conf. Meas. Tech. and Mechatr. Autom.*, vol. 3, 2009, pp. 374–379, doi: 10.1109/ICMTMA.2009.422.

[9] M. Zhang, Y. Lu, and T. Xia, "The Design and Implementation of Virtual Machine System in Embedded SoftPLC System," in *Int. Conf. Computer Sci. and Appl.*, 2013, pp. 775–778, doi: 10.1109/CSA.2013.185.

[10] M. Simros, M. Wollschlaeger, and S. Theurich, "Programming embedded devices in IEC 61131-languages with industrial PLC tools using PLCopen XML," in *CONTROLO'2012 Portug. Conf. Autom. Control*, 2012, pp. 51–56.

[11] S. Cavalieri, G. Puglisi, M.S. Scroppo, and L. Galvagno, "Moving IEC 61131-3 applications to a computing framework based on CLR Virtual Machine," in *IEEE 21st Int. Conf. Emerg. Techn. Fact. Autom.*, 2016, pp. 1–8, doi: 10.1109/ETFA.2016.7733632.

[12] Y. Lee, J. Jeong, and Y. Son, "Design and implementation of the secure compiler and virtual machine for developing secure IoT services," *Future Generation Computer Systems*, vol. 76, pp. 350–357, 2017, doi: 10.1016/j.future.2016.03.014.

[13] M. Okabe, "Development of processor directly executing IEC 61131-3 language," in *SICE'08 Ann. Conf.*, 2008, pp. 2215–2218, doi: 10.1109/SICE.2008.4655032.

[14] P. Mazur, R. Czerwinski, and M. Chmiel, "PLC implementation in the form of a System-on-a-Chip," *Bull. Pol. Acad. Sci. Tech. Sci.*, vol. 68, no. 6, pp. 1263–1273, 2020, doi: 10.24425/bpasts.2020.135386.

[15] *LLVM Compiler Infrastructure*. [Online]. Available: www.llvm.org (Accessed 2021-11-24).

[16] T. Catalão and M. de Sousa, "IEC 61131-3 Front-End for the LLVM Compiler Family," in *25th Int. Conf. Emerg. Techn. Fact. Autom.*, vol. 1, 2020, pp. 1191–1194, doi: 10.1109/ETFA46521.2020.9211921.

[17] D. Rzońca, J. Sadolewski, A. Stec, Z. Świder, B. Trybus, and L. Trybus, "Mini-DCS system programming in IEC 61131-3 Structured Text," *JAMRIS*, vol. 2, no. 3, pp. 48–54, 2008.

[18] D. Rzońca, J. Sadolewski, A. Stec, Z. Świder, B. Trybus, and L. Trybus, "Developing a multiplatform control environment," *JAMRIS*, vol. 13, no. 4, p. 73–84, 2019, doi: 10.14313/JAMRIS/4-2019/40.

[19] B. Trybus, "Development and Implementation of IEC 61131-3 Virtual Machine," *Theoret. Appl. Informatics*, vol. 23, no. 1, pp. 21–35, 2011.

[20] *CPDev VM public sources*. [Online]. Available: https://github.com/CPDev-ControlProgramDeveloper (Accessed 2021-11-24).

[21] K.R. Irvine, *Assembly language for X86 processors (Eigth edition)*. Hoboken: Pearson, 2018.

[22] F.A. Turbak and D.K. Gifford, *Design Concepts in Programming Languages*. The MIT Press, 2008.

[23] K.D. Cooper and L. Torczon, *Engineering a Compiler (Second Edition)*. Boston: Morgan Kaufmann, 2012.

[24] E. Ferreira, R. Paulo, D. Cruz, and P. Henriques, "Integration of the ST Language in a Model-Based Engineering Environment for Control Systems – An Approach for Compiler Implementation," *Comp. Sci. and Inform. Sys.*, vol. 5, no. 2, pp. 87–101, 2008, doi: 10.2298/CSIS0802087F.

[25] *Lumel S.A.* [Online]. Available: www.lumel.com.pl (Accessed 2021-11-24).

[26] *Instrument Science Systems*. [Online]. Available: www.issi.com.ph (Accessed 2021-11-24).

[27] *Praxis Automation Technology*. [Online]. Available: www.praxis-automation.nl (Accessed 2021-11-24).

[28] *iGrid T&D*. [Online]. Available: www.igrid-td.com (Accessed 2021-11-24).