GRAPHICAL APPLICATION FOR SIMULATING THE OPERATION OF A MICROPROCESSOR

C.M. Muscai, *Babeș-Bolyai University, Reșița, ROMANIA*
M.D. Stroia*(*Corresponding author)***,** *Babeș-Bolyai University, Reșița, ROMANIA*
C. Hațiegan, *Babeș-Bolyai University, Reșița, ROMANIA*
C.Popescu, *"Constantin Brâncuşi" University of Târgu Jiu, ROMANIA*

ABSTRACT: This paper presents a software application designed to simulate the basic operation of a microprocessor for educational purposes. The application employs a custom, easy-to-use assembly language and provides an integrated environment for both programming and simulation of the proposed microprocessor. As such, it serves as a valuable didactic tool for introducing students to assembly programming concepts and to the fundamental principles governing the operation of microprocessors and microcontrollers. The application was developed using LiveCode, a free programming language chosen for its capability to compile applications across major operating systems, including both desktop and mobile platforms. Consequently, the program can be executed on any computer without requiring external libraries, installation, or elevated user privileges. The compiled files are fully portable and can run on any system based on Windows, Linux, or macOS.

KEY WORDS: microcontroller, programming, design, education.

1. TECHNOLOGICAL CONTEXT AND MOTIVATION

The rapid evolution of technology, particularly in the field of mechatronics, has led to an almost complete abstraction of hardware components. Within this modern approach, most functionalities are handled exclusively through external software libraries, whose internal mechanisms are often not fully understood. This abstraction has increased the difficulty of comprehending the actual operation of hardware components.

The use of modern educational tools and development platforms such as Arduino or Raspberry Pi significantly enhances the overall quality of teaching, especially in the area of microcontrollers. Several studies confirm that this approach benefits students by increasing motivation and employability while also improving programming skills and facilitating individual learning [1].

Computational thinking has become an essential requirement, driven by the growing process of digitalization and the increasing demand for engineers skilled in programming microcontrollers and IoT devices, which are now ubiquitous in both industrial and domestic environments [2].

Mechatronic applications developed exclusively through high-level libraries tend to be large and resource-intensive, consuming considerable storage and RAM space on microcontrollers [4] . Although the processing power of modern devices has improved dramatically, achieving high energy efficiency and execution speed still requires a deep and precise understanding of the underlying principles and specific characteristics of mechatronic devices [5].

Currently, several types of code interpreters and just-in-time compilers are used for microcontroller programming, each offering distinct advantages and disadvantages in terms of memory footprint and execution speed [5].
This paper proposes a new perspective on microcontroller programming by introducing a didactic software application designed to be simple, intuitive, and easy to manage for both students and educators.

2. SIMULATOR DESIGN AND IMPLEMENTATION

The Small CPU Emulator was designed as an educational tool with a clear and minimalist interface, focusing strictly on functionality relevant to learning assembly-like programming concepts.
The design of the minimal processor implemented in the Small CPU Emulator application was inspired by the educational computer Know-how Computer, a concept created in the 1980s by Wolfgang Back and Ulrich Rohde. The original system was conceived as a paper-based learning tool, where users manually wrote programs and traced their execution step by step using a pen. In its initial form, the system featured eight memory registers and twenty-one possible lines of code, while the register values were represented visually using matchsticks placed within squares, each symbolizing a register [4].
The complete list of instructions implemented by the virtual processor is presented in Table 1, along with a brief description of each instruction's function. The virtual processor integrated into the application adopts a simplified structure compared to the original concept. It consists of two working registers R1 and R2 and a third register, RM, used for storing auxiliary data or intermediate states during program execution.
To preserve simplicity, the instruction set was minimized as follows:

- 2 arithmetic instructions,
- 2 register testing instructions,
- 3 data transfer instructions (for register-to-register copying),
- 3 input/output control instructions, plus:
- NOP (No Operation),
- STP (Stop Program – terminates execution),
- JMP (Jump – branches to a specified line).

To better illustrate the execution stages of a program, the emulator includes the option to run the virtual processor either in step-by-step mode or in automatic mode, with an adjustable execution speed controlled via a graphical slider interface.

Table 1. The complete list of the virtual CPU instructions

Instruction	Type	Action
INC Rx	Arithmetical	Adds 1 to the value of R1 or R2 register
DEC Rx	Arithmetical	Subtracts 1 from the value of R1 or R2 register
NOP	Generic	No action, just delay
STP	Generic	Marker for end of the program
JMP xx	Control	Makes the row xx the next operation to run, unconditional jump
REX	Register	Swaps the content of the two registers R1 and R2
STO Rx	Register	Stores the value in R1 or R2 into RM register
LOD Rx	Register	Restores, loads the value stored in the RM register into R1 or R2
ISZ Rx	Comparison	Compares the value stored in R1 or R2 and jumps over the next instruction if condition is met
ISE	Comparison	Compares the values stored in R1 and R2, if they are equal then it jumps over the next instruction

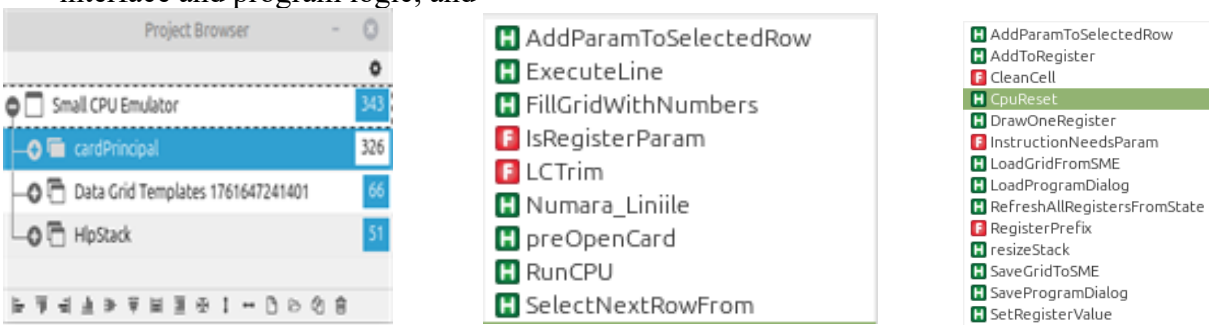
2.1. Small CPU Emulator in LiveCode

Applications developed in LiveCode are stored as binary stacks that include both executable code and all interface objects (such as buttons, fields, and images) as well as optional graphical or audio assets.
Internally, each LiveCode project can contain one or more stacks, each composed of one or

more cards where interface elements are placed.
These cards can be shown or hidden dynamically during execution, allowing transitions, animations, or modular interface behavior.
The implemented application, as shown in figure 1.a, contains:

- a main stack named "Small CPU Emulator", which hosts the main interface and program logic, and
- a secondary stack, "*HlpStack*", which contains the help window accessible through the main menu.

Each object within the stack contains associated handlers and functions. The number of lines of code for each object is indicated in Figure 1.b and Figure 1.c.



a) architecture b) handlers and functions in Card c) handlers and functions in Stack

Figure 1. Small CPU Emulator structure

2.2. Small CPU Emulator coding

The application initialization begins in the handler *preOpenCard*, executed when the *CardPrincipal* card is loaded.

This routine expands the main menu to fit the entire window width and initializes the DataGrid object *StringGridComputer* with 50 numbered, empty rows, using *FillGridWithNumbers*, as shown in code sequence from figure 2.a.

```
command FillGridWithNumbers pGridName
  local tDataA, i

  repeat with i = 1 to 50
    put i into tDataA[i]["Nr"] -- numbered line index
    put empty into tDataA[i]["Sel"]
    -- Instruction mnemonic column ("Comanda")
    put empty into tDataA[i]["Comanda"]
    -- Parameter / operand column ("Param")
    put empty into tDataA[i]["Param"]
  end repeat

  set the dgData of group pGridName to tDataA
  send "SelectRow 1" to group "StringGridComputer" in
1 millisecond

end FillGridWithNumbers
```

a) FillGridWithNumbers coding

```
-- SaveGridToSME pFilePath
--  line 1: KRY_M_GRIDFILE_v1
--  subsequent lines: tab-separated values:
Nr<TAB>Comanda<TAB>Param

command SaveGridToSME pFilePath
  local tOut, tLine,tTotal, tRow,tRowA,tTab

  put numToChar(9) into tTab
  put kSMESignature & cr into tOut -- Start with
signature
  put the dgNumberOfLines of group
"StringGridComputer" into tTotal
  repeat with tRow = 1 to tTotal
    put the dgDataOfLine[tRow] of group
"StringGridComputer" into tRowA
    put CleanCell(tRowA["Nr"]) into tLine
    put tLine & tTab & CleanCell(tRowA["Comanda"])
into tLine
    put tLine & tTab & CleanCell(tRowA["Param"]) into
tLine
    put tLine & cr after tOut
  end repeat
  put tOut into url ("file:" & pFilePath)
  answer "Saved succesfully !"
end SaveGridToSME
```

b) SaveGridToSME coding

Figure 2. Small CPU Emulator coding 1

The simulator allows manual editing or file-based loading of assembly-like programs. Saving functionality is handled by the *SaveGridToSME* command, shown in figure 2.b, which generates a *.sme file containing the current instruction set. The file header

includes a unique signature to prevent loading incompatible file types.
Before writing data to file, the system cleans all strings to remove invalid characters or trailing spaces using the function *CleanCell* form figure 3.a.
The core execution routine, *ExecuteLine* (figure 3.b), receives the line number of the instruction to execute. It reads the command and parameter fields, validates them, and then executes the corresponding operation. If the instruction is empty, execution skips to the next line; otherwise, it executes the appropriate case in the switch–break block.

```
-----------------------------------------------------------------
-- CleanCell
-- Remove leading/trailing spaces and tabs
-----------------------------------------------------------------
function CleanCell pText
  local t, tLen
  put pText into t

  -- strip leading spaces/tabs
  repeat while (t begins with " ") or (t begins with
numToChar(9))
    delete char 1 of t
  end repeat

  -- strip trailing spaces/tabs
  put length(t) into tLen
  repeat while tLen > 0 and (char tLen of t is " " or char
tLen of t is numToChar(9))
    delete char tLen of t
    subtract 1 from tLen
  end repeat

  return t
end CleanCell
```

a) CleanCell coding

```
-----------------------------------------------------------------
-- Execute ONE instruction from the currently selected
row (gSelectedRow)
-----------------------------------------------------------------
command ExecuteLine pRow
  local tRowA, tOpcode, tOpcodeUC, tParamText
  local tDidMove
  put false into tDidMove
  -- get row data
  put the dgDataOfLine[pRow] of group
"StringGridComputer" into tRowA
  put LCTrim(tRowA["Comanda"]) into tOpcode
  put LCTrim(tRowA["Param"]) into tParamText
  put toUpper(tOpcode) into tOpcodeUC
  if tOpcodeUC is empty then
    SelectNextRowFrom pRow
    exit ExecuteLine
  end if
  switch tOpcodeUC
    case "INC"
      if IsRegisterParam(tParamText) then
        AddToRegister tParamText, 1
      else
        answer "INC expects R1 or R2."
      end if
      break
```

b) ExecuteLine coding

Figure 3. Small CPU Emulator coding 2

This command works in tandem with helper routines such as *IsRegisterParam()* which validates operand type and *AddToRegister()*, responsible for modifying the register values dynamically. A separate command, *RunCpu*, manages program execution in automatic mode, handling delays, UI updates, and stopping conditions for cases as infinite loops.

2.3. Operating Small CPU Emulator

The interface was designed for simplicity and clarity. A DataGrid element displays the current program instructions, each row containing the line number, command, and parameter. Instruction insertion is performed via graphical buttons representing predefined commands, eliminating the need to memorize syntax or mnemonics.
Each button is accompanied by a short descriptive label. The application also includes:

- three control buttons for Run, Stop, and Step-by-step execution;
- visual representations of registers using *PictureBox* controls, each showing an incandescent lamp image that toggles based on the register's value;
- increment/decrement buttons for each register, allowing real-time modification even during program execution.

The program state can be saved or loaded using the File → Open/Save menu. Files are stored in plain ASCII *.sme format with TAB-separated columns, allowing easy inspection or manual editing under any OS.
When launched, the application loads a blank instruction list with registers reset, mimicking a cold start or hardware reset.
Navigation is possible via mouse or keyboard arrows. A visual arrow cursor marks the current line being executed or edited.
Instructions are inserted using the on-screen buttons, as one can notice from figure 4, while parameters can be added to selected lines.
The editor prevents invalid parameter insertions but detects missing operands only at

runtime.In such cases, the emulator halts execution and notifies the user of the error.
The JMP instruction is handled through a dialog that allows selecting the target line number. The entered value is automatically displayed in the parameter column.
The automatic execution mode *RunCpu* manages sequential instruction processing, observing user-defined speed, and allowing interruption or manual continuation from any desired line.

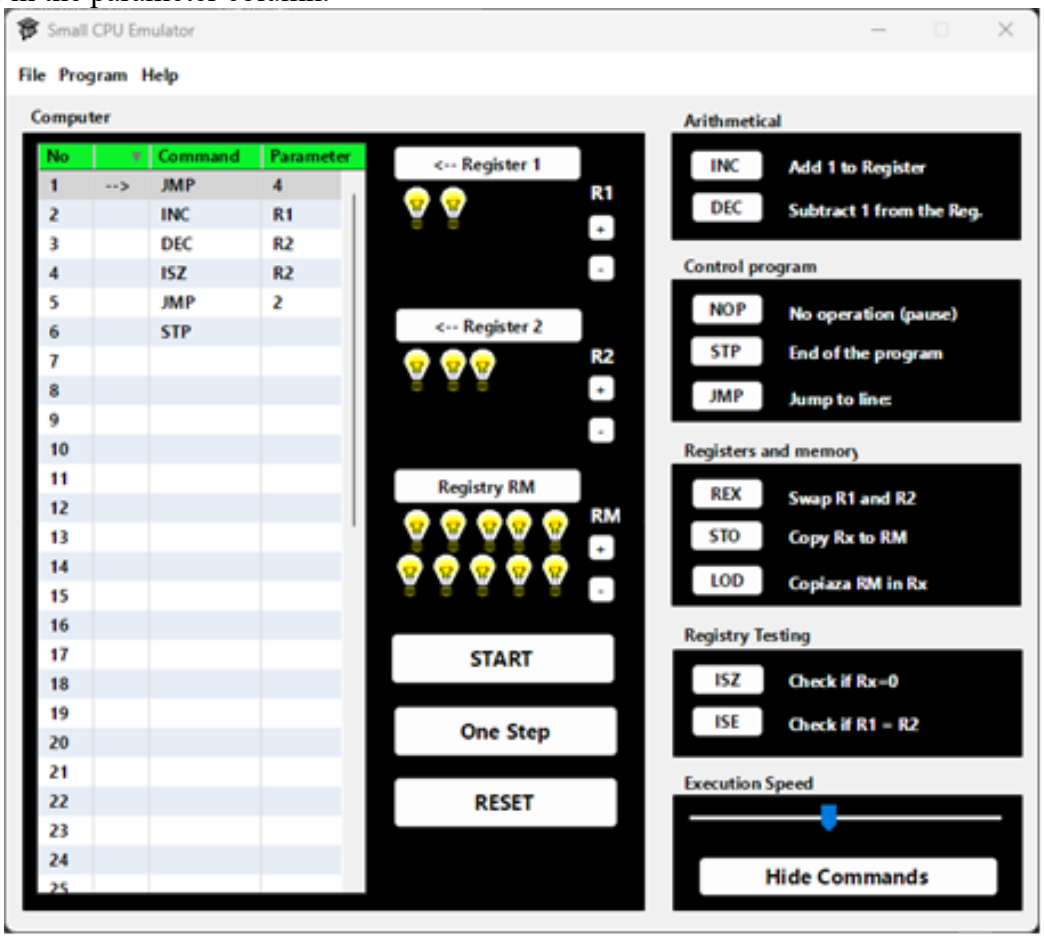


Figure 4. Using the Small CPU Emulator

3. CONCLUSION

The Small CPU Emulator provides a lightweight and intuitive environment for simulating basic assembly-like programs.
Its graphical simplicity encourages experimentation and learning, helping students grasp microprocessor fundamentals without the overhead of complex toolchains.
Despite its limited instruction set, the emulator supports a wide range of programmatic scenarios and serves as an effective didactic resource.
Future developments will aim to extend the instruction set with more advanced arithmetic operations, implement memory addressing, and introduce dual decimal/binary visual representation of data — features commonly found in professional or web-based emulators.

REFERENCES

[1] Abdiakhmetova, Z., Temirbekova, Z., Aimal Rasa, G., Berdaly, A. Using of microcontroller for student learning process. Journal of Mathematics, Mechanics and Computer Science, 122(2), 114–123, 2024.
[2] Fülöp, M.T.; Udvaros, J.; Gubán, Á.; Sándor, Á. Development of Computational Thinking Using Microcontrollers Integrated into OOP. Sustainability 2022

[3] Little Man Computer https://en.wikipedia.org/wiki/Little_Man_Computer
[4] Visan I., Diaconu "Home Automation System Using ESP8266 Microcontroller and Blynk Application[J]". The Scientific Bulletin of Electrical Engineering Faculty, 21(2):59-62, 2021.
[5] Vrbančič, F., Kocijančič, S. Strategy for learning microcontroller programming—a graphical or a textual start?. Educ Inf Technol 29, 5115–5137, 2024
[6] WDR paper computer https://en.wikipedia.org/wiki/WDR_paper_computer