Operating System Simulator to Translate Assembler Code to Machine Code

Enrique Ayala[1], Francisco A. Madera[2], Luis Basto[3]
Universidad Autónoma de Yucatán, Facultad de Matemáticas
Periférico Norte Tablaje 13615, Mérida, Yucatán, México

Abstract. *We analysed and implemented an Operating System software simulator to translate assembler code into machine code. The simulator is a computer program of a virtual machine which contains a computer simplified computer architecture and a memory manage module. The simulator allows users to implement routines in order to extend several functionalities such as memory management, control processing, and any other Operating System role. The simulator can also be employed to make different practices to help in the computers architecture understanding.*

Keywords — *Assembler Language, Machine Language, Operating Systems, Compilers, Translators.*

I. Introduction

Assembler language and its conversion to the machine code is a complex action since it requires the knowledge of the computer architecture and the set of input operations. To generate the output, it needs the language analysis, translation's construction, lexical and syntactical elements identification. The loading, execution, and programming of the Operating System (OS) involves the knowledge of the processor features such as process concept, instruction set, and memory management.

Nowadays, the lack of assembler language programming in computer sciences academic programs is notoriously. These academic programs prefer to teach subjects such as systems programming, operating systems and compilers which require the understanding of the OS and the architecture processor.

We construct a software simulator to help in the learning of the assembler language, avoiding hardware considerations that gives the majority of the problems. In particular, the instruction set must be known, according to the processor employed. We could use this simulator to learn assembler in any personal computer disregarding the type of processor.

Systems programming allows to create base software to interact with the computer, accessing directly to the sources and devices of the machine to act as a service platform for the users and applications, in such a way that the system management is simplified. The Operating System manages and provides mechanisms to access all the hardware resources such as processor, memory, hard disk, etc. Compilers deals with the programming language tracking to create an executable file. The literature is not enough for several concepts to be appropriately understood and assimilated, more tools are needed to achieve the required abstraction level.

In this work, we describe the implementation of a software simulator to identify and learn the phases and components involved from the compilation and the machine code generation to load and visualize a process into memory. The main goal is to show the translation of the assembler code to the machine code, step by step, showing every phase in depth.

We analyse and implement a computer program, a simulator of a virtual machine which contains a simplified computer architecture and a memory manage module of the Operating System. Additionally, a translator program was constructed to have a source code in assembler language to be translated to machine code.

II. Related Work

The usage of simulators in education as a didactical tool has generated successful cases due to students are able to make several practices to understand the concepts. Software simulators can be found in several areas such as biology, medicine, mechanic, management, etc.

In Computer Sciences, simulators are employed to analyse computer inner process and control the time, manipulate variables, and provide programming tools to users in order to increase their knowledge according to their individually requirements and learning.

Leland [1] describes SIC (Simplified Instructional Computer), a hypothetical machine to abstract essential functions of a real system to help in the systems programming learning by using a simplified architecture. It also provides the architecture description and the instruction set, emulators and assemblers written in Pascal programming language.

Deitel and Deitel [2] proposed a virtual machine development and its machine language called SML (Simpletron Machine Language). This architecture defines a 4-decimal digit form, the first 2 digits for

the operation code and the last two digits for the memory addressing. It employs a reduced instruction set that work with a pair of records and a limited memory of 100 words.

Lopes et al. [3] presented a proposal, for the development of educational software simulator, which has visual and interactive graphics, serving as a teaching tool for both professors and students. The simulator makes a representation of the functions, related to memory management through a graphics scene. However this software did not help in the abstraction necessary to understand how operating systems work.

Cahya [4] proposes a simulator that consists of a computer system hardware and kernels which are presented as a software. The CPU, memory, clock, and Input/Output operations are simulated as a software. The architecture of the simulator being discussed has three modules: virtual machine (Operating System) Kernels and Simulated Computer Systems. Students can practice with their own design and implementation, according to the modules the simulator provides. It lacks a compiler or translator, so it does not allow to write a code and generate the machine code for a memory visualization.

Saraswat and Gupta [5] designed and implemented a process scheduler simulator, which is focused on evaluating the suitability of various Process scheduling algorithms for a Multimedia Operating System such as: First Come First Serve (FCFS), Multi-Level Feedback (MLFS), Shortest Job First (SJF) and Earliest Deadline First (EDL). Its objective was to compare and calculate several metrics with such algorithms.

The afore mentioned proposals are limited for the description and development of emulators to execute machine code. The user interface is available in text mode o with a graphical user interface but with few options to interact with the virtual machine. They do not define an OS core to manage the execution programs and systems, for instance, the multiprogramming. They neither have options to manipulate variables to control the simulation process.

III. The Operating System

An OS is the software that supports the computer's basic functions due to it manages hardware resources. It is responsible for handling a number of important functions such as the Central Process Unit (CPU), memory management, input/output devices, security, networking, devices, and file systems, among others. OS can be programmed to access the hardware resources (Figure 1). This kind of programming is called system programming (SP), and aims to produce software which provides services to the computer hardware. SP requires greater degree of hardware awareness and also involves compilers, linkers, macros, device drivers, networking.

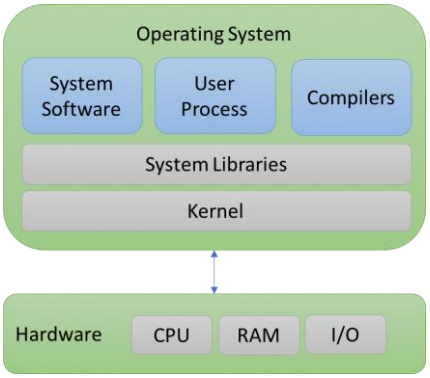

Fig. 1. The Computer hardware and user applications interface.

Computers understand the binary (machine) language and perform basic operations millions of times per second: add, subtract, memory data movement, etc. To extent these basic operations, there are many programming languages classified according to the abstraction level as shown in Figure 2.

High Level Languages (C, C++, Fortran, and Java) are the most common Programming Languages since they are intuitive for humans. An assembly language (ASM) is the most basic programming language available for any processor, the code works only with operations that are implemented directly on the physical CPU.

Assembly languages generally lack high-level conveniences such as variables and functions, and they are not portable between various families of processors. They have the same structure and set of commands as machine language, but it allows the programmer to use names instead of numbers. This language is still useful for programmers when speed is necessary or when they need to carry out an operation that is not possible in high-level languages.

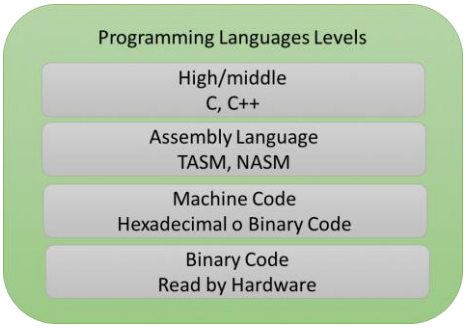

Fig. 2. The programming language levels.

Machine Languages (ML) consist of binary numbers recognizable by the CPU. The CPU architectures can be RISC (Reduced Instruction Set Computer) or CISC (Complex Instruction Set Computer). Every processor type has its own set of specific machine instructions which can be a data processing, data transfer and flow control. Memory holds ML programs and data, the CPU fetches ML instructions from memory and executes them (Figure 3).

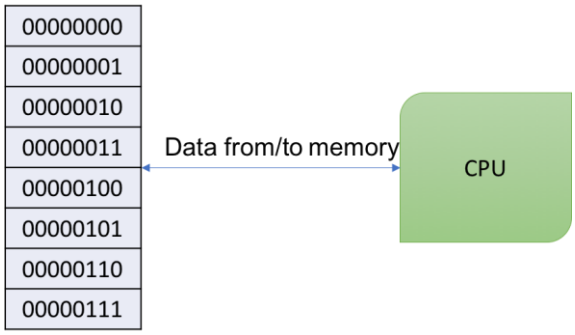

Fig. 3 The CPU fetches ML instructions from memory.

ML instructions are made up of several fields, the Opcode and the Operands. The Opcode stands for operation code and it specifies the operation to be performed. The Operand fields indicates where to obtain the source and destination operands for the operation specified by the opcode.

IV. The Architecture

Popular assemblers have emerged over the years for the Intel family of processors: TASM (Turbo Assembler from Borland), NASM (Netwide Assembler for both Windows and Linux), and GNU (Assembler distributed by the free software foundation). A linker program is required to produce executable files. Debuggers allow to trace the execution of the program, and visualizes code, memory, and registers.

The Instruction Set Architecture (ISA) is a collection of assembly/machine instruction set that can be managed with memory instructions and programmer-accessible registers. Computers have three main components interconnected with buses: processor, memory and input/output devices. A bus serves to transfer data, transfer addresses or for control. A processor consists of an ALU (Arithmetic Logic Unit), registers, control unit; and its programming varies from one processor to another. The memory can be referenced by an ordered sequence of bytes. The physical address space is determined by the address bus width, for instance, the Pentium has a 32-bit address bus.

Assembler	Machine	
#read name, age #S1, S2 are messages	@163	Memory usage equals the sum of the number of lines, the integer variables, string variables times 40, and 25 additional spaces for the stack program 16 + 2 + (3x40) + 25 = 163
ASP 60	80060	Integer value 60 is placed in register A
STA M	210016	Register A is stored in memory (maximum age)
RS S1	130017	Read string into S1 (message 1)
RS S2	130057	Read string into S2 (message 2)
RS N	130097	Read string in to N (variable name)
RD E	100137	Read integer value into E (age)
LDA M	200016	Load register A from memory (variable M)
COMP E	440137	Compare E (age) to M (maximum age)
JGT J1	430012	Jump if register A is greater than 0 to label J1
WS N	140097	Write variable N (name)
WS S1	140017	Write variable S1 (message 1)
END	640000	End Program
J1		Label J1
WS N	140097	Write variable N
WS S2	140057	Write variable S2
End	640000	End program

Fig. 4. An example from ASM to ML: age classification.

The simulator proposed in this work utilizes an architecture that is a variation of the model [1] and [2]. The instruction word is formed by 6 decimal digits, the first two digits for the operation code and the last four digits for the memory address. The maximum addressing memory store up to 10,000 words. Variables are integer and the string data has 40 characters.

An example of the ASM – ML is illustrated in Figure 4. Memory usage is 16 + 2 + (3x40) + 25 = 163; there are 16 ASM instructions, 2 integer variables (E, M), string variables (S1, S2, N). This indicates that 163 bytes are held to create the processes. The first instruction is placed on the position memory 0. Since the maximum memory is 10,000, then an address is labelled with 4 digits.

V. The Simulator

The Prototype Model [6] was chosen for the implementation, and consists on the software development while having the functional elements visible. Additionally, this is feedback, tested, and adjusted according to the user requirements.

C++ programming language and Qt Creator IDE were employed for the simulator construction. These are handy for a friendly graphic user interface and contains data structures and algorithms templates. The simulator was implemented in a PC, an Intel Duo Core Processor 2.16 GHz , 4GB RAM. Basic instructions are needed to run ML programs: ASM-ML translator, and operative system to allow the interaction with end users. Aside from this, the simulator provides tools to the users to add modules and algorithms. The simulator design is depicted in Figure 5.

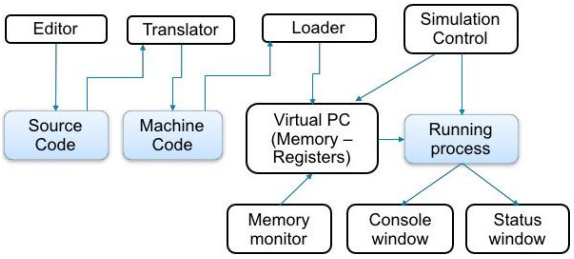

Fig. 5. Simulator architecture design.

Simulator modules are the user interface, ASM to ML translator, memory usage, and multiprocessing handler. The user interface has two main options, program generation and simulation. ASM programs are loaded or created, so that they can be edited (Figure 6).

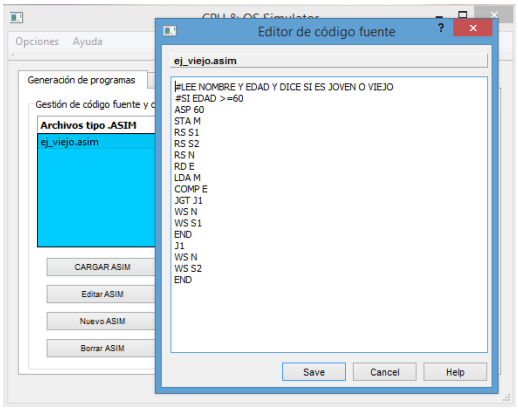

Fig. 6. Editing ASIM files.

These files are translated to ML, by using two phases in order to identify the instruction memory addresses (Figure 7).

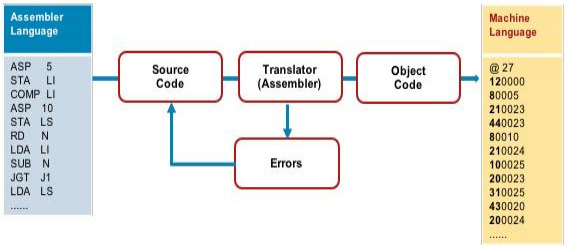

Fig. 7. ASM to ML translation.

If the translation is performed successfully, a ML file is generated, otherwise an error message is displayed. The ML files generated are ready to be used to create processes and then to be run. A process is enabled with the Start Simulation button (Figure 8).

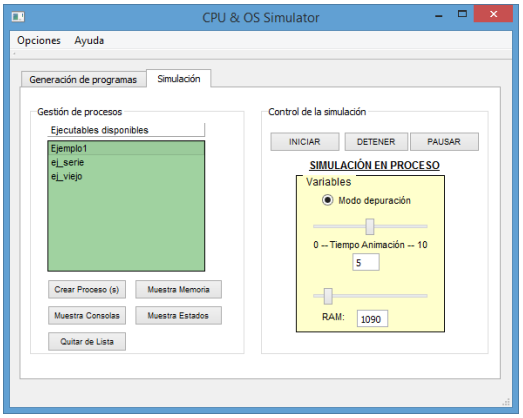

Fig. 8. The ML files are ready to start the simulation process.

There is a status window where the information displayed, taken from the PCB (Processes Control Block), is related to the instructions on memory, process status and registers (Figure 9). The PCB structure stores the process identifier, state, size of process memory, begin and end of space memory, memory pointer for instruction on execution, stack pointer, accumulator register, and current instruction of ML file.

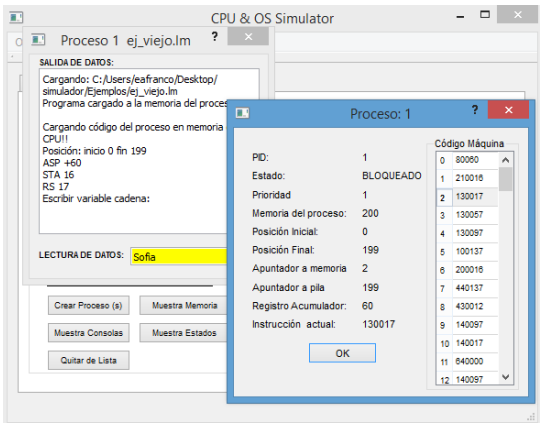

Fig. 9. The status windows of a running process.

Figure 10 depicts two running processes, displaying the information about the executed instruction, the assembler mnemonic and the number indicates the variable in real memory address. If the depuration mode is unmarked, on the Simulation tab, then the assembler instructions are hided at window console and only information of input and output instructions will be shown.

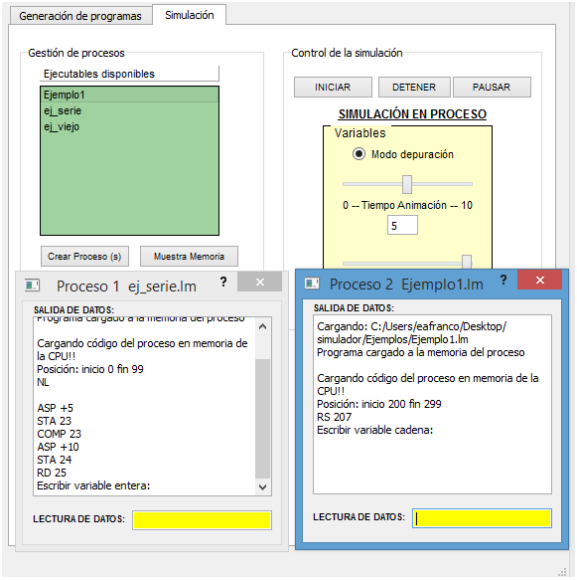

Fig. 10. Two running processes and their output data, in debugger or depuration mode.

Programs are loaded to memory in a partitioned allocation [7] using a contiguous memory assign algorithm with fixed partition in every process created; afterwards a PCB structure is added to a list. This list is controlled according to a Round-Robin management [8], where each process has a lapse of time called quantum and is assigned to the processor in a counter clock wise order. A process can change its status (New, Ready, Running, Locked, Finalized) and the information is displayed in the process console (Figure 11).

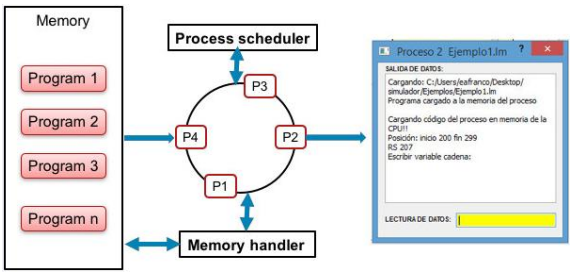

Fig. 11. The processes management

Figure 12 illustrates the OS interchange process, changing the state of one of such running processes to allow executing, restoring registers of CPU, and changing the state of the other process to be locked. It also saves all registers into its PCB, following the principles of Time Sharing Systems and Multitasking [9].

Memory monitor is activated with the "Show Memory" button and displays information of the memory status. It also provides options to remove processes. In Figure 13 the processes with Id 1 and Id 2 are in locked status, the memory partition for ej_serie.lm image program begins at position 0 and ends at position 99.

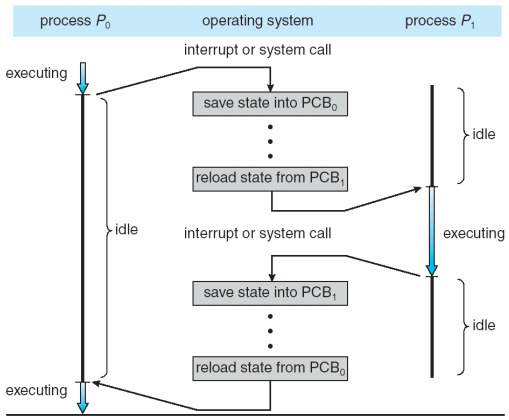

Fig. 12. The switching process [10]

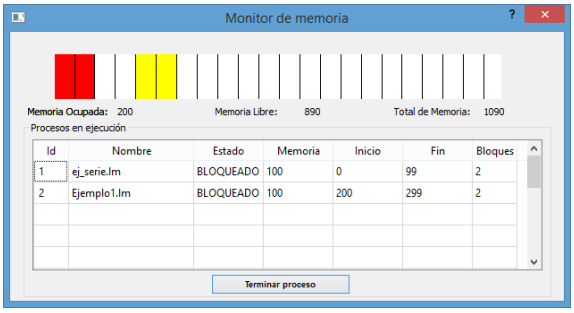

Fig. 13. Memory monitor window.

VI. Conclusion

The design and implementation of a software simulator is presented. This simulator translates ASM code to Machine Code using the core of an operating system with multiprocessing. ASM and ML files can be loaded, created and edited. The memory management allows to visualize the busy memory and allows to release during the running time. Many processes can be activated at the same time, making the memory management to arrange the instructions and variables in the memory. This kind of simulators is not easy to find in the literature, so that the design of the program simulator could serve as the basis for an extended simulator with more functionalities.

The simulator software was used as a didactical tool in the base programming subject for the computer science bachelor program of the Yucatan University. Students implemented some routines for memory management, processes management, running and input times, in order to test the performance of the operations. Scheduling routines were implemented: First Come First Serve (FCFS), Shortest-Job-First (SJF) scheduling, Priority scheduling, random choosing. Other routines were

also implemented to assign memory to the processes depending on the availability: the first memory room, the best memory room size, the worst memory room size.

REFERENCES

[1] B. Leland. "Software de Sistemas. Introducción a la Programación de Sistemas". Addison Wesley, 1988.

[2] H. Deitel, P. Deitel, Como Programar en C/C++ y Java. 4ª Edición, Pearson, 2004.

[3] Á. R. Lopes, D. A. de Souza, J. R. B. de Carvalho, W. O. Silva and V. L. P. de Sousa, "SIME: Memory simulator for the teaching of operating systems," *2012 International Symposium on Computers in Education (SIIE)*, Andorra la Vella, pp. 1-5, 2012.

[4] S. Cahya, "Designing Operating System Simulator: A Learning Tool," *2009 11th International Conference on Computer Modelling and Simulation*, Cambridge, pp. 156-160, 2009.
doi: 10.1109/UKSIM.2009.92

[5] P. K. Saraswat and P. Gupta, "Design and Implementation of a Process Scheduler Simulator and an Improved Process Scheduling Algorithm for Multimedia Operating Systems," *2006 International Conference on Advanced Computing and Communications*, Surathkal, pp. 513-517, 2006.

[6] R. Pressman. Ingeniería de software un enfoque práctico. México: McGraw-Hill, 2005.

[7] Andrew, Tanenbaum, "Sistemas Operativos Modernos". 3ª Ed. México, 2009.

[8] M. Barrionuevo, M. F. Piccoli, R. Apolloni. Una herramienta de Simulación para la Planificación de Procesos. Revista Iberoamericana de Educación en Tecnología y Tecnología en Educación. No. 9. Abril 2013.

[9] A. Silverschatz, P. Galvin, G. Gagne. "Operating System Concepts. 9th Ed", 2013.

[10] (2017), Gitbook.textbook of Operating System [Online]. Available:
https://www.gitbook.com/book/ayushverma8/test-book/details