Revista Brasileira de Ensino de Ciência e Tecnologia
ISSN: 1982-873X

https://periodicos.utfpr.edu.br/rbect

A simulator to support assembly language teaching

Rene Pegoraro
rene.pegoraro@unesp.br
0000-0003-0314-8660
Universidade Estadual Paulista, Bauru, São Paulo, Brasil.

Marcelo Nicoletti Franchin
marcelo.franchin@unesp.br
0000-0003-3021-9874
Universidade Estadual Paulista, Bauru, São Paulo, Brasil.

ABSTRACT

This article describes a simulator of a simple hypothetical processor used for introducing assembly language concepts to high school and university students. The simulator, developed to be used as a didactic tool, offers to students, in a graphical interface, a model of how a computer works from the point of view of low-level programming. In this tool, users load the program in machine language and visualize the changes resulting from its execution in the processor's memory and registers. Considering the importance of understanding the difference between assembly language and machine language, students are instructed to write their code in assembly language and then to obtain the machine language. The assembly process begins manually and then it is done through an assembler program. Manual assembly helps to explain some concepts related to the generation of executable code hidden in integrated development environments. Although the tool simulates a simple hypothetical processor, it was built following the instruction syntax used in Intel's 32-bit architecture (IA-32), allowing students to use the concepts learned to understand other assembly languages on real computers. This tool is used at the beginning of the Assembly Language course in the Computer Science Program at Sao Paulo State University, located in Bauru/SP/BR. Data collected over eight years, four of them using the simulator, suggest the pass rate of students has increased significantly.

KEYWORDS: Assembly Language. Simulator. Computing Teaching.

INTRODUCTION

Most students and professionals prefer technological tools that facilitate and speed up development work in the field of computing. Oftentimes, however, they ignore the internal structure and functioning of these technologies, which have the potential to boost the use of such tools. Thus, despite this preference for high-level programming, specialized libraries, human-machine interfaces, and data abstraction mechanisms, as well as the emphasis on understanding computers as machines represented by abstract models, assembly language remains important for it is where many of the mechanisms within those tools are built. Assembly language appears in several courses such as: i) Introduction to Computing, relating high-level language with assembly language and machine language, data encoding and subroutine’s call and return mechanisms; ii) Compilers, transforming high-level programming languages into assembly or machine language; iii) Computer Architecture and Organization, starting from the organization to the execution of the machine language in the architecture; and iv) Operating Systems, in situations that require low-level instructions, programmed with assembly language, in which it is not possible to obtain high execution performance when coding in high-level languages, such as process context switching or service calls of the system. This language also appears in the curriculum guides of the Association for Computing Machinery (ACM) for undergraduate courses in Computer Science (ACM/IEEE, 2013) and Computer Engineering (ACM/IEEE, 2016), which reinforces its relevance.

The Assembly Language course is usually taught in the second or third term of Computing Programs, after a term in which students learn programming in some high-level language. Despite the complexity of this course, it should not discourage beginner students in the field of computing.

In this sense, programming tools can be used to simplify learning, encourage, and motivate students. Although tools such as integrated programming environments with visualization of registers and processor memory sections, microcontroller programming, and simulators and emulators of complex or hypothetical real devices are used in teaching, they tend not to focus on the simplicity of understanding the functioning of desirable low-level processor programming concepts in a teaching tool.

This article, then, introduces AS.SIM (ASsembly SIMulator), used to teach students the fundamentals of assembly language and machine language programming. It consists of four eight-bit registers, two indicators (flags), three control buttons and 256 memory locations, all of them presented simultaneously in a single fixed window, configuring a simplified model of a processor.

After this Introduction, the article is organized as follows: Section 2 presents a review of some related simulators used for teaching assembly language; section 3 describes the characteristics of the AS.SIM simulator and its use; section 4 shows the approach used in the Computer Science program at UNESP Bauru; section 5 indicates the results obtained with the use of this tool; and the conclusion section offers some final remarks.

TOOLS FOR TEACHING ASSEMBLY LANGUAGE

A vast array of tools can be used to teach assembly language, ranging from environments aimed at professional development and testing of programs on real processors, to hypothetical processor simulators for teaching computer architecture and assembly language.

With a focus on teaching assembly language, it is expected that the tool will allow students with little experience to learn a simple architecture that is the basis for programming a real processor as well as provide a simple interpretation of the data being presented. In addition, the tool is expected to offer ease of use, availability in operating systems common to students of computing courses, and visualization of a step-by-step execution of instructions in a program. Complex environments with multiple windows and unnecessary information for beginner students can lead them to try to understand information that is not relevant for that stage, diverting their attention and making it difficult to learn the content at hand. In this sense, a simple and clear interface that presents nothing but memory, registers and primary flags, encourages students to focus on understanding low-level programming. Thus, for beginners, professional development environments are discarded in order for simulators with simpler architectures to remain.

It is highly desirable that the knowledge acquired at the beginning of the course not be lost when students advance their learning, the used architecture be introductory to one of the most common architectures in the program in which students are inserted, and the tool accompany part of a real existing architecture. Therefore, for AS.SIM, we opted for 32-bit Intel Architecture (IA-32) (INTEL, 2016).

A study conducted by Nikolic, Radivojevic and Djordjevic (2009) comparing 28 simulators shows that almost all of them make programming in machine language possible; however, the didactics of these simulators in programming assembly language are not addressed. Another study, this time by Esmeraldo *et al*. (2019), compares 16 simulators using seven metrics, including one specific for assembly language. Of these, the CompSim, MarieSim, SIMAEAC, SimuS were chosen for beginner students in view of their simplicity.

CompSim (ESMERALDO; LISBOA, 2017) includes a graphical interface integrated to a simulated hardware platform, which allows to simplify the configuration of hardware components, processor programming, and execution and performance evaluation of new computer systems. In the low-level programming window, one can see the CPU's internal registers, the cache memory, part of the main memory, and the assembly language code. Some registers can be accessed through memory locations. Indirect addressing to memory happens only by STI and LDI instructions, in which a memory location indicates the location to be manipulated. There are only two types of jumps, JN (jump if negative) and JZ (jump if zero).

MARIE, that is, Machine Architecture that is Really Intuitive and Easy, was developed to understand the basics of a fully functional von Neumann architecture with an uncomplicated set of instructions. MARIE is the basis for the MarieSim simulator. All system components, including registers, instructions and memory, are simultaneously visible on the screen. The 13 instructions are composed of two bytes fixed size and are specific to MARIE. In the main window, one can see the CPU's internal registers, part of the 4k main memory, and the assembly language code. Unlike many conventional architectures, it does not use flags and conditional jumps consider the accumulator value to be less than, equal to or greater than zero. (NULL; LOBUR, 2003).

SIMAEAC, i.e. Academic Simulator for Teaching Computer Architecture, implements the 8085's processor architecture, through the following types of instructions: logical, arithmetic, stack manipulation, subroutine call and return, conditional and unconditional branches, and data transfer between registers and between registers and memory. It displays registers A (Accumulator), B, C, D, E, H, L, PC and SP, all composed of 8 bits, and 5-bit flags. A reduced memory area, from 00 to 8F of addressing, is fully presented in the interface without the need to scroll the sidebar. As a result, the visualization of the cell in which the instruction being executed is found is always available. With this feature, instructions that handle memory, in addition to the SP and PC registers, must handle 8 bits. (VERONA; MARTINI; GONÇALVES, 2009).

Silva and Borges (2016) present the SimuS simulator and the hypothetical Sapiens processor, which is the evolution of the Neander-X processor that presents 31 instructions with up to three bytes each. Some of the main improvements, among many, are the 64K of memory, the indirect addressing, the stack and specific instructions for calling and returning subroutines. Its interface is simple: the main screen presents an area for typing the assembly language; buttons to run, step by step and stop; the registers; the flags; part of the 64k of memory; and the entry and exit. The loading of the program into the memory happens directly through assembling the code typed in assembly language. Weber (2001 *apud* SILVA; BORGES, 2016) define the Neander machine as a rudimentary architecture based on an accumulator with 256 bytes of RAM, instructions with up to two bytes, direct modes, and 16 instructions.

Besides the cited research, robust literature focused on specific simulators for the teaching and learning process can be found. Some of them are quoted in the present section.

Silva and Borges (2018) present the architecture of a didactic processor and a simulator that interacts with sensors and actuators, with the goal of introducing concepts of IoT (Internet of Things) in the courses of assembly language and computer architecture. The simulator can be run on a *Raspberry Pi Nano* computer in which students can read and control sensors and actuators connected to general purpose (*GPIO*) pins directly from the code executed by the simulator. The approach is to learn introductory reading and writing issues in sensors and actuators without the complexities of real systems.

Research by Sartor *et al*. (2020) is focused on a tool for the study of computer architecture in a simplified microprocessor called uPD, based on VHDL and with a small instruction set. Questionnaires were administered to students who provided useful feedback for improving the tool. It is used in several courses of the computer engineering program.

A low complexity processor (LCP) was created to be simple enough to be designed and implemented in the time allocated to laboratory classes (KOSTADINOV; BENCHEVA, 2019). The LCP processor is a Harvard architecture machine, with 8-bit accumulator, 256x12-bit program memory and 256-byte data memory. Two 8-bit ports provide communication with the outside world and a minimal instruction set with only two addressing modes (absolute and immediate) exist. The assembler was also created to run the assembly language programs. Results indicate that students were extra motivated as a consequence of their successfully designed processor, which could run programs quickly. A few students stated that they presented relative difficulty and needed help.

Research by Santa *et al*. (2017) indicates that the concepts students need to learn about processor theory are obtained from, at least, a medium complexity processor, which is always quite complex for beginner students. To make it easier to get started in the world of processors, we propose a project with a great reduction of components from an already small real processor. The minimalist approach makes understanding hardware and assembly language easier, improving the learning process and encouraging students to study processors.

In their work with the CompSim simulator, Esmeraldo *et al*. (2020) reiterate that the use of simulators and virtual environments are complementary pedagogical practices. Simulators are important tools to the knowledge appropriation process because they foster the development of skills and practical experiences, asynchronously, in virtual scenarios that resemble reality. Simulators are also critical in laboratories that lack infrastructure or when there is a need to reduce costs, perform quick setups, and get instant results. CompSim's hardware platform is Mandacaru. It has a 16-bit processor model with cache and RAM memories, system and peripheral buses.

WebRISC-V is a didactic simulator recently developed by Mariotti and Giorgi (2020) that supports 64-bit RISC-V instruction set and processors. The simulator supports students in studying and investigating the reasons for the good performance of pipeline processors and allows them to easily examine the processors' basic architectural elements as well as their internal states. The tool is highly complex and is not intended to be used with beginner students.

The LC-3 simulator, *Little Computer* 3, mimics the LC-3 hypothetical architecture as defined by Patt and Patel (2004). It defines a bank constituted by eight general purpose registers, 16 instructions with various address modes, three Z, N and P flags. The call to subroutines is not conventional and does not explicitly use stack. If necessary, it should be implemented by the programmer.

The J1 simulator is based on SAP-1, *Simple As Possible* (MALVINO, 1985). Simple as it is, the simulator has 16 memory positions and 16 instructions, including a conditional jump statement. It has no stack, subroutine call instructions or indirect addressing (MANSOUR; NAVIN; RAHMANI, 2013).

GNUSim8085[1] is a graphic simulator, assembler and debugger for the Intel 8085 microprocessor capable of operating on Linux and Windows. This simulator supports all of 8085's instructions. A code written in assembly language can be assembled and directly charged to the simulator, in a clear way to users, allowing debugging directly from the code typed in assembly language.

AS.SIM presents ideas that are similar to the analyzed simulators. SIMAEAC (VERONA; MARTINI; GONÇALVES, 2009) has all 144 processor memory positions visible at the same time. MarieSim (NULL; LOBUR, 2003) offers the advantages of using an accumulating architecture so as to contribute to beginner students' learning. Many of the original simulators or simulator evolutions are known to share common features such as indirect addressing, stack, specific instructions for subroutines and number representation in hexadecimal, decimal and binary systems. In addition to the characteristics cited above, students' comprehension of the mechanics of the manual assembly of a program and the possibility of automatic assembly by an assembler program, which is offered separately by AS.SIM, are taken into consideration.

The diverse contributions by different researchers presented in this article are deemed important to the development of microprocessor teaching, both in hardware and software. However, the design requirements for each of them are different, and only a few are aimed at beginner students learning assembly language. Using 32-bit or 64-bit RISC processor simulators makes the learning process more challenging and time-consuming. At the same time, the tools are aimed at detailed architectures with many elements, which can make the learning process even harder. The approach used in this research tries to speed up the learning process by working with a simple architecture, but at the same time with essential elements found in most existing architectures.

THE SIMULATOR

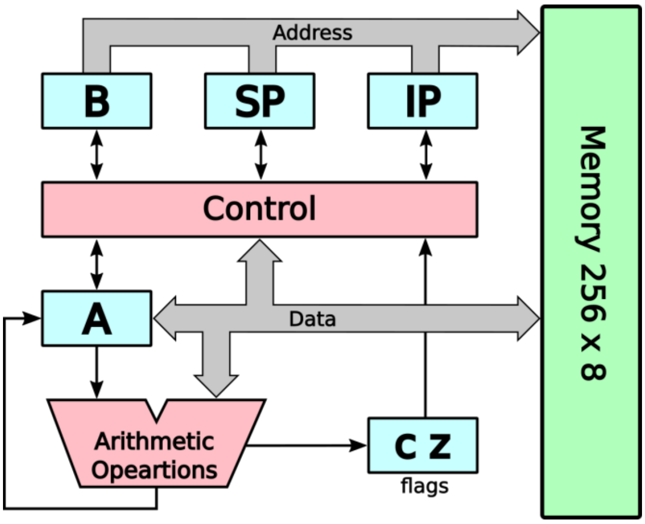
AS.SIM was specifically designed for the Assembly Language course offered by the Computer Sciences program at the Sao Paulo State University, located in Bauru, Brazil. The goals include simplifying the understanding of programming in assembly language and machine language. It was developed in *Processing*[2] because it runs on Windows, Linux and Mac. When an application is exported using *Processing*, the source codes accompany the application, reinforcing the free software premise.

The development of AS.SIM was fostered by the need for a pedagogical tool in the form of a simulator with a simplified architecture, which would allow beginner students to create a mental model of a computer, before they could proceed to learn more complex programming in assembly language. To answer to this demand, AS.SIM has an interface window that contains all the minimum elements required for learning the fundamentals of programming in assembly language.

Despite its simplicity, this simulator operates based on the very same fundamental concepts of current computers with von Neumann architecture. The architecture simulated by AS.SIM represents a computer with 16 basic instructions. The arithmetic operations can be used with immediate, direct and indirect addressing by register. The 256 memory positions are used for code, data and stack. The architecture operates with no signal integers, thus the C and Z indicators are sufficient to control conditional branches. Nevertheless, arithmetic operations with and without signal can be performed naturally if the complement of two is considered for values greater than 127.

Its architecture is composed of four 8-bit registers. It is based on accumulator, that is, one of the data in arithmetic operations always comes from the accumulator, the register A, and the other data comes from one of the available address modes. When the operation comes to an end, the result is returned to the accumulator register. Register B is intended for indirect access operations to memory, and the value contained in it indicates the memory location that will be used as one of the operands. The architecture also has two control registers: the IP register (instruction pointer), which indicates the memory location where the following instruction is to be performed, and the SP register (stack pointer), which is used to maintain the stack that is the basis for the rescue of the return address positions of the subroutines. The two flags Carry and Zero indicate the overflow or zero in arithmetic operations. Abstract representation of the architecture is presented in Figure 1.

Figure 1 – AS.SIM Architecture Model



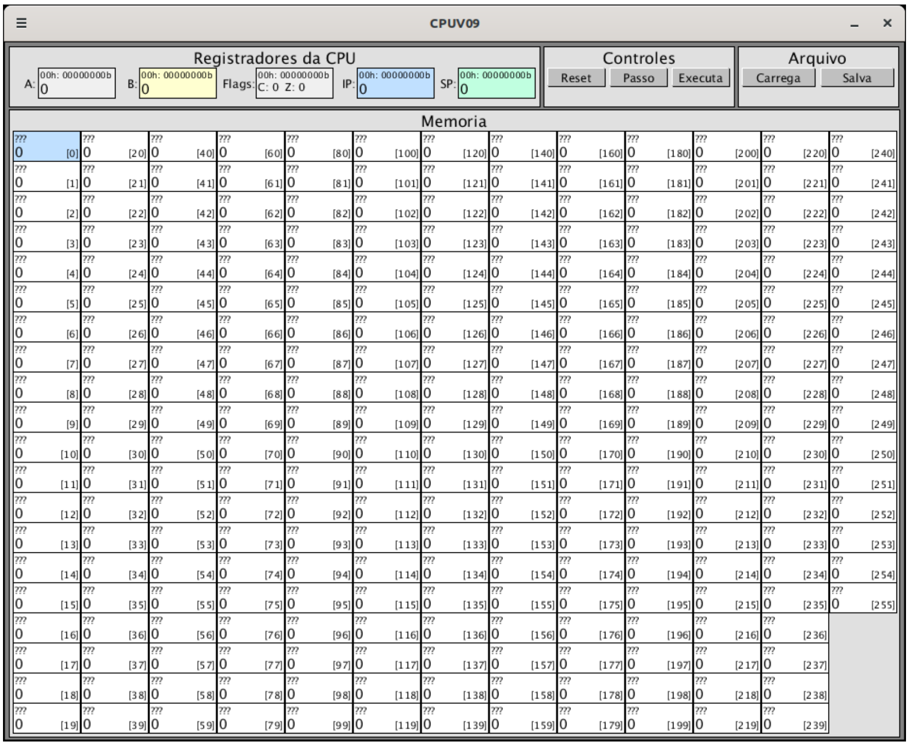
Source: Authors (2020).

Instructions can be composed of one or two bytes. The first byte always indicates the opcode - the instruction code to be executed. In two-byte instructions, the second byte indicates an argument of the instruction.

User Interface

With the intention of building a mental model of a computer in students' minds, AS.SIM presents all simulated structures in a single window, without menus or configurations. In this window all memory locations and registers are represented. In the same window, there are five buttons for controlling the simulation, loading and saving the values stored in memory in a file. This window is shown in Figure 2.

Figure 2 – Simulator Window



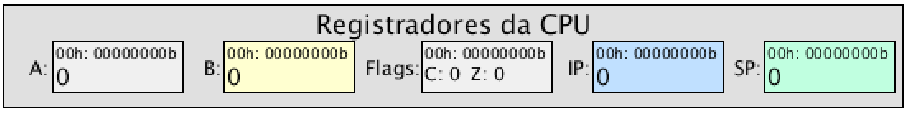
A: 00h: 00000000b 0
B: 00h: 00000000b 0
Flags: 00h: 00000000b C: 0 Z: 0
IP: 00h: 00000000b 0
SP: 00h: 00000000b 0

??? 0 [0]	??? 0 [20]	??? 0 [40]	??? 0 [60]	??? 0 [80]	??? 0 [100]	??? 0 [120]	??? 0 [140]	??? 0 [160]	??? 0 [180]	??? 0 [200]	??? 0 [220]	??? 0 [240]
??? 0 [1]	??? 0 [21]	??? 0 [41]	??? 0 [61]	??? 0 [81]	??? 0 [101]	??? 0 [121]	??? 0 [141]	??? 0 [161]	??? 0 [181]	??? 0 [201]	??? 0 [221]	??? 0 [241]
??? 0 [2]	??? 0 [22]	??? 0 [42]	??? 0 [62]	??? 0 [82]	??? 0 [102]	??? 0 [122]	??? 0 [142]	??? 0 [162]	??? 0 [182]	??? 0 [202]	??? 0 [222]	??? 0 [242]
??? 0 [3]	??? 0 [23]	??? 0 [43]	??? 0 [63]	??? 0 [83]	??? 0 [103]	??? 0 [123]	??? 0 [143]	??? 0 [163]	??? 0 [183]	??? 0 [203]	??? 0 [223]	??? 0 [243]
??? 0 [4]	??? 0 [24]	??? 0 [44]	??? 0 [64]	??? 0 [84]	??? 0 [104]	??? 0 [124]	??? 0 [144]	??? 0 [164]	??? 0 [184]	??? 0 [204]	??? 0 [224]	??? 0 [244]
??? 0 [5]	??? 0 [25]	??? 0 [45]	??? 0 [65]	??? 0 [85]	??? 0 [105]	??? 0 [125]	??? 0 [145]	??? 0 [165]	??? 0 [185]	??? 0 [205]	??? 0 [225]	??? 0 [245]
??? 0 [6]	??? 0 [26]	??? 0 [46]	??? 0 [66]	??? 0 [86]	??? 0 [106]	??? 0 [126]	??? 0 [146]	??? 0 [166]	??? 0 [186]	??? 0 [206]	??? 0 [226]	??? 0 [246]
??? 0 [7]	??? 0 [27]	??? 0 [47]	??? 0 [67]	??? 0 [87]	??? 0 [107]	??? 0 [127]	??? 0 [147]	??? 0 [167]	??? 0 [187]	??? 0 [207]	??? 0 [227]	??? 0 [247]
??? 0 [8]	??? 0 [28]	??? 0 [48]	??? 0 [68]	??? 0 [88]	??? 0 [108]	??? 0 [128]	??? 0 [148]	??? 0 [168]	??? 0 [188]	??? 0 [208]	??? 0 [228]	??? 0 [248]
??? 0 [9]	??? 0 [29]	??? 0 [49]	??? 0 [69]	??? 0 [89]	??? 0 [109]	??? 0 [129]	??? 0 [149]	??? 0 [169]	??? 0 [189]	??? 0 [209]	??? 0 [229]	??? 0 [249]
??? 0 [10]	??? 0 [30]	??? 0 [50]	??? 0 [70]	??? 0 [90]	??? 0 [110]	??? 0 [130]	??? 0 [150]	??? 0 [170]	??? 0 [190]	??? 0 [210]	??? 0 [230]	??? 0 [250]
??? 0 [11]	??? 0 [31]	??? 0 [51]	??? 0 [71]	??? 0 [91]	??? 0 [111]	??? 0 [131]	??? 0 [151]	??? 0 [171]	??? 0 [191]	??? 0 [211]	??? 0 [231]	??? 0 [251]
??? 0 [12]	??? 0 [32]	??? 0 [52]	??? 0 [72]	??? 0 [92]	??? 0 [112]	??? 0 [132]	??? 0 [152]	??? 0 [172]	??? 0 [192]	??? 0 [212]	??? 0 [232]	??? 0 [252]
??? 0 [13]	??? 0 [33]	??? 0 [53]	??? 0 [73]	??? 0 [93]	??? 0 [113]	??? 0 [133]	??? 0 [153]	??? 0 [173]	??? 0 [193]	??? 0 [213]	??? 0 [233]	??? 0 [253]
??? 0 [14]	??? 0 [34]	??? 0 [54]	??? 0 [74]	??? 0 [94]	??? 0 [114]	??? 0 [134]	??? 0 [154]	??? 0 [174]	??? 0 [194]	??? 0 [214]	??? 0 [234]	??? 0 [254]
??? 0 [15]	??? 0 [35]	??? 0 [55]	??? 0 [75]	??? 0 [95]	??? 0 [115]	??? 0 [135]	??? 0 [155]	??? 0 [175]	??? 0 [195]	??? 0 [215]	??? 0 [235]	??? 0 [255]
??? 0 [16]	??? 0 [36]	??? 0 [56]	??? 0 [76]	??? 0 [96]	??? 0 [116]	??? 0 [136]	??? 0 [156]	??? 0 [176]	??? 0 [196]	??? 0 [216]	??? 0 [236]	
??? 0 [17]	??? 0 [37]	??? 0 [57]	??? 0 [77]	??? 0 [97]	??? 0 [117]	??? 0 [137]	??? 0 [157]	??? 0 [177]	??? 0 [197]	??? 0 [217]	??? 0 [237]	
??? 0 [18]	??? 0 [38]	??? 0 [58]	??? 0 [78]	??? 0 [98]	??? 0 [118]	??? 0 [138]	??? 0 [158]	??? 0 [178]	??? 0 [198]	??? 0 [218]	??? 0 [238]	
??? 0 [19]	??? 0 [39]	??? 0 [59]	??? 0 [79]	??? 0 [99]	??? 0 [119]	??? 0 [139]	??? 0 [159]	??? 0 [179]	??? 0 [199]	??? 0 [219]	??? 0 [239]	

Source: Authors (2020).

The registers appear at the top of the interface and have their values presented in decimal, hexadecimal and binary representation (Figure 3). As in a real computer, the registers cannot be edited directly by the users, but they are guaranteed to have the IP register and the SP set to zero by pressing the Reset button.

Figure 3 – CPU's Registers and Flags



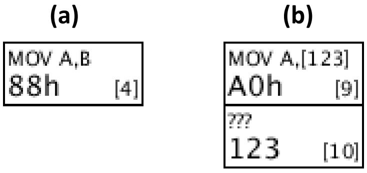
A: 00h: 00000000b 0
B: 00h: 00000000b 0
Flags: 00h: 00000000b C: 0 Z: 0
IP: 00h: 00000000b 0
SP: 00h: 00000000b 0

Source: Authors (2020).

The memory cells have 8 bits and their presentations are as shown in Figure 4. In each cell the stored value is presented and, if there is an instruction with the corresponding opcode, the mnemonic instruction is indicated. In the lower right corner, there is a fixed cell position number in square brackets, which helps students find the desired cell and get the notation used in Intel's IA-32 syntax to indicate a memory location. All 256 memory locations are presented at the same time, providing a general view to users.

Figure 4a shows memory cell 4, which stores the 88h value. It is possible to note that the 88h opcode in this cell corresponds to the instruction "MOV A,B", an instruction of one byte only. Figure 4b shows memory cells 9 and 10, with opcode A0h and data 123. As it is a two-byte instruction, in position 9, the instruction shown is "MOV A,[123]", where address 123 comes from the argument value given in the following memory cell, cell 10.

Figure 4 – Memory cells



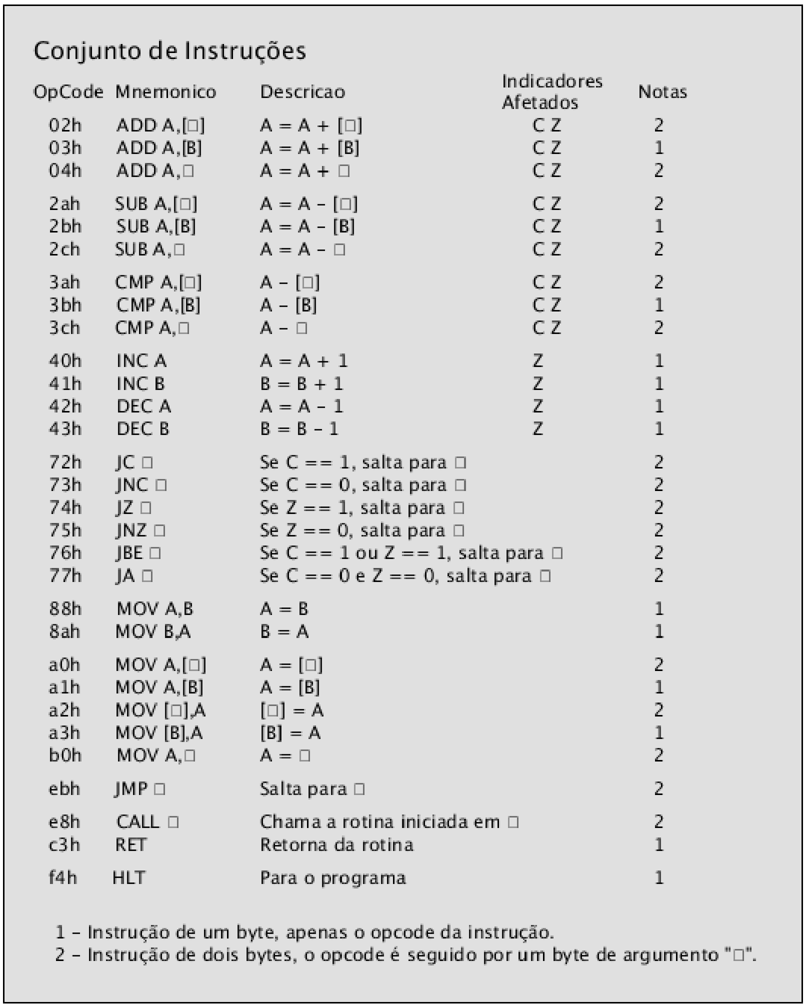
Source: Authors (2020).

The user interface also features five buttons, two for loading and saving files in machine language and three for controlling the simulator's execution. The three control buttons are 1) the **Reset** button, which restarts the processor resetting the IP registers – program counter – and the SP – stack pointer – ; 2) the **Passo** button, which only runs the instruction indicated by the IP, repositioning it and allowing the users to run the program instruction by instruction; and 3) the **Executa** button, which sequentially runs the program's instructions, advancing the IP and showing changes in memory and registers at a rate of two instructions per second. During the execution of the program, started by pressing the **Executa** button, the label of this button changes to **Para**, suggesting the possibility of stopping the program at the current execution position.

Trying to keep the user interface as simple as possible, AS.SIM has the main window and just another help window indicating useful information about the instructions available in the simulator (see Figure 5). AS.SIM's help window encourages the use of the syntax of Intel instructions used by assemblers for the IA-32, which helps students, in a second moment in the course, program a real computer without the impact of learning new mnemonics for the new architecture.

The help window presents a table where each line corresponds to an instruction and the columns contain the following information: i) instruction opcode; ii) mnemonic; iii) a brief description of the operation in a similar format to that used in high-level languages; iv) the flags indicators affected by the instruction; and v) a note on the number of bytes of the instruction. The symbol "□" represents where the value of the second byte that constitutes the instruction will be inserted. The square brackets "[" and "]" represent direct or indirect addressing with registers, so that the memory cell in the indicated position between them will be used by the instruction.

Figure 5 – AS.SIM help window



Conjunto de Instruções

OpCode	Mnemonico	Descricao	Indicadores Afetados	Notas
02h	ADD A,[□]	A = A + [□]	C Z	2
03h	ADD A,[B]	A = A + [B]	C Z	1
04h	ADD A,□	A = A + □	C Z	2
2ah	SUB A,[□]	A = A - [□]	C Z	2
2bh	SUB A,[B]	A = A - [B]	C Z	1
2ch	SUB A,□	A = A - □	C Z	2
3ah	CMP A,[□]	A - [□]	C Z	2
3bh	CMP A,[B]	A - [B]	C Z	1
3ch	CMP A,□	A - □	C Z	2
40h	INC A	A = A + 1	Z	1
41h	INC B	B = B + 1	Z	1
42h	DEC A	A = A - 1	Z	1
43h	DEC B	B = B - 1	Z	1
72h	JC □	Se C == 1, salta para □		2
73h	JNC □	Se C == 0, salta para □		2
74h	JZ □	Se Z == 1, salta para □		2
75h	JNZ □	Se Z == 0, salta para □		2
76h	JBE □	Se C == 1 ou Z == 1, salta para □		2
77h	JA □	Se C == 0 e Z == 0, salta para □		2
88h	MOV A,B	A = B		1
8ah	MOV B,A	B = A		1
a0h	MOV A,[□]	A = [□]		2
a1h	MOV A,[B]	A = [B]		1
a2h	MOV [□],A	[□] = A		2
a3h	MOV [B],A	[B] = A		1
b0h	MOV A,□	A = □		2
ebh	JMP □	Salta para □		2
e8h	CALL □	Chama a rotina iniciada em □		2
c3h	RET	Retorna da rotina		1
f4h	HLT	Para o programa		1

1 – Instrução de um byte, apenas o opcode da instrução.
2 – Instrução de dois bytes, o opcode é seguido por um byte de argumento "□".

Source: Authors (2020).

AS.SIM operation

Simulations in AS.SIM are executed from instructions loaded in memory, and this can be done directly by the users, either by typing the opcodes of the instructions and their data in the memory cells or by loading an object file with the data to be assigned to the memory.

For users to insert a program into the memory cells, they need to understand the principles of assembly, that is, the transformation from assembly language to machine language. Thus, they write their program in assembly language using the mnemonics and data of each instruction. Then, they do the assembly manually, which consists of transforming the mnemonics to the corresponding opcode bytes, generating a sequence of opcodes and data. Finally, they insert this sequence, byte and byte, into the simulator's memory cells. The assembly must comply with the available instructions, presented in the help window, as shown in Figure 5.

The object file used by AS.SIM is a text file containing up to 256 lines, each line refering to the corresponding memory cell. This file can be either edited by the users in a text editor or generated by an assembler program from an assembly language source file. An assembler program automates the transformation of an assembly language file to a machine language file, the latter being called an object file. Both in the object file and in the opcodes and typed data, each value can be represented in decimal or hexadecimal when followed by an "h".

AS.SIM operates with jumps and calls to subroutines with absolute destinations, as they are more intuitive and facilitate the calculation of destinations in the assembly. Conditional or fixed jump instructions, when at the time of their branches, cause the second byte of the instruction - the destination address - to be loaded into IP. Similarly, when a CALL instruction is used, IP+2 is saved in the stack and IP is loaded with the address of the subroutine. When executing the RET instruction, the value taken from the stack is assigned to the IP register, returning from the subroutine.

In IA-32, the conditional jump instructions are divided into two groups: signed and unsigned conditional jumps, with signed operations using signal and overflow indicators in addition to the C and Z indicators (INTEL, 2016). Since AS.SIM is composed of only the C and Z indicators, the conditional jump instructions always operate from the affected indicators in the arithmetic instructions on unsigned integers, and follow the IA-32 mnemonics nomenclature for the applied conditional branches to unsigned integers. The six conditional jump instructions, when used with the SUB or CMP instruction, cover the conditions of lesser (JC), greater (JA), equal (JZ), different (JNZ), less than or equal (JBE) and greater than or equal (JNC). Instructions JC, JNC, JZ, JNZ check one indicator only to decide on the branch; instructions JA and JBE depend on both indicators, where JA checks that C and Z are off and JBE checks that C or Z are on.

USE OF AS.SIM IN THE ASSEMBLY LANGUAGE COURSE

AS.SIM has been used in the Assembly Language course since 2014. This course is offered once a year, consists of a 2-hour class a week, and takes place during the second term of the Computer Sciences Program. AS.SIM is an introductory tool employed in the first 12 hours of this course.

An integrated environment for the development of professional programs may hide steps or generate misunderstandings on how a program is assembled and loaded in memory. With AS.SIM, students are encouraged to firstly write their program in assembly language without any directive, and then perform the assembly manually and test their programs. As beginner students initially need to write the program directly into assembly language, relations between machine, assembly and high-level languages are reinforced, which deepens the concepts of computing fundamentals.

Once students have acquired the concept of assembly, directives and labels for constant values and memory locations are added so that they can start using an assembler program. At the same time, students are encouraged to develop their own assembly program under teacher guidance in relation to the assembler's specifications, directives, instructions, addressing modes and to the syntax normally found in commercial assemblers. The teacher also instructs that students need to perform two steps to complete their assembly. The first step is to determine the position of the labels used in the branch and subroutine destinations, and the second step is to deal with the assembly of instructions and writing the machine language in the object file. The proposed assembler, in addition to the instructions available and indicated in the help window (see Figure 5), offers the ORG, EQU, DB and END directives.

Once students have understood the functioning of a CPU's architecture and its programming in machine and assembly language, they are taught about another tool, with a professional integrated development environment, which offers compiling, assembling, loading, executing and debugging the code with the visualization of registers, memory sections and flags.

RESULTS

AS.SIM has been used in the Computer Science Program at the Faculty of Sciences at Unesp in Bauru since 2014, which allows for a quantitative analysis based on the pass rate of the Assembly Language course.

Data were collected from students who took and passed the Assembly Language course between 2011 and 2018. During this period, 325 students enrolled in the course, including those who dropped out and those who failed and had to take the course in the following year. The proportion of those who passed are shown in Table 1. AS.SIM was used by students in 2014, 2015, 2016 and 2018. In 2017, a substitute teacher taught the course and did not use AS.SIM.

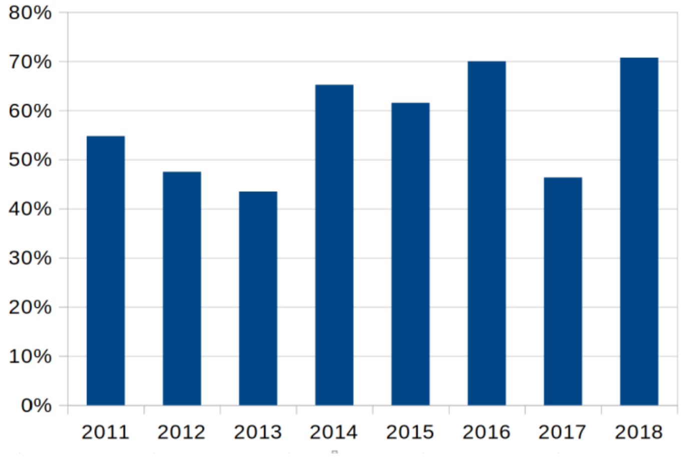
Table 1 – Passing Students in Each Year

Year	Total number of students	Passing students	Pass rate (%)
2011	42	23	55%
2012	40	19	48%
2013	46	20	43%
2014	46	30	65%
2015	39	24	62%
2016	30	21	70%
2017	41	19	46%
2018	41	29	71%

Source: Authors (2020).

Analyzing the graph of students' pass rates, shown in Figure 6, built from the last line of Table 1, data suggest that in the years 2014, 2015, 2016 and 2018, when the simulator was used, the pass rate was greater than or equal to 62%, whereas this same rate was less than 55% during the other years. Ignoring other adverse factors that may have influenced the pass rate, the improvement shown by students who used AS.SIM seems to be significant.

Figure 6 – Percentage of Passing Students per Year



Source: Authors (2020).

Table 2 shows the distribution of the number of students in the course divided by final grade ranges, with each range having a width of 1.0 and the final grades assigned to students ranging from 0.0 to 10.0. Students who dropped out are in the range of [0, 1]. In this table, students are divided into classes that used the simulator, with 156 students; and classes that did not use the simulator, with 169 students.

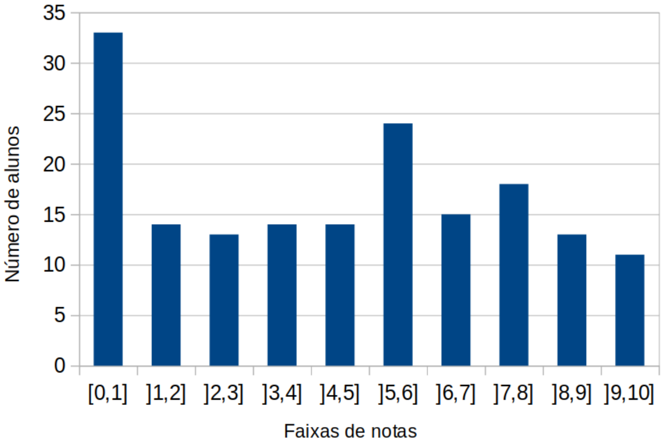
Table 2 – Grade Distribution by Range

Grades	With simulator	Without simulator
[0, 1]	2	17
]1, 2]	7	13
]2, 3]	16	14
]3, 4]	5	13
]4, 5]	13	23
]5, 6]	22	15
]6, 7]	26	16
]7, 8]	19	19
]8, 9]	20	14
]9,10]	14	8

Source: Authors (2020).

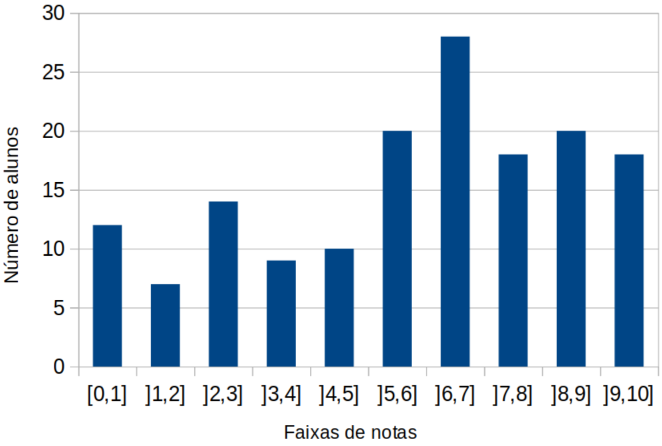
Comparing the graphical representations in Figure 7 and Figure 8 and Table 2, one can observe that the range of students with grades [0, 1], who did not use the simulator, in Figure 7, is preponderant. On the other hand, the highest concentration of grades of students who used AS.SIM, Figure 8, is in the range]6, 7]. This observation demonstrates that the use of a pedagogical tool well adapted to the taught content can encourage students, reducing evasion and improving their performance in the course.

Figure 7 – Students' Grade Distribution in Classes that did not use AS.SIM



Source: Authors (2020).

Figure 8 – Students' Grade Distribution in Classes that used AS.SIM



Source: Authors (2020).

Students report that the direct execution of the simulator without the need for installation, its user-friendly interface and, above all, the simplicity of the simulated CPU architecture helped them to understand the concepts of assembly language.

CONCLUSIONS

AS.SIM is a proposal for a simple simulator based on the von Neumann model to start teaching Assembly Language. For the teacher, it is an option of a pedagogical tool among other simulators with similar goals. The adoption of AS.SIM can be justified by the ease of explaining the simple features of this simulator, as it is easy to use and install in most common operating systems used in personal computers.

Due to the simplicity of AS.SIM, students can focus on learning the assembly language and its conversion to machine language. The use of the simulator does not demand extra effort from students, as it only requires them to load the opcodes and data into memory and press the **Executa** button to test their program.

Data obtained from 2011 to 2018 in teaching the course Assembly Language at Unesp, Bauru, indicate some advantages in using this simulator. These data show that the use of a pedagogical tool, well adapted to the teaching domain, can improve students' achievement.

UM SIMULADOR DE APOIO AO ENSINO DE LINGUAGEM DE MONTAGEM

RESUMO

Este artigo descreve um simulador de um processador hipotético simples para a introdução de conceitos de linguagem de montagem em alunos do segundo grau e nível superior. O simulador, desenvolvido para ser utilizado como uma ferramenta didática, oferece ao aluno, em uma interface gráfica, um modelo do funcionamento de um computador do ponto de vista da programação de baixo nível. Nesta ferramenta, o usuário carrega o seu programa em linguagem de máquina e visualiza as mudanças decorrentes da sua execução na memória e nos registradores. Considerando a importância do entendimento da diferença entre linguagem de montagem e linguagem de máquina, o aluno é orientado a escrever seu código em linguagem de montagem e em seguida, fazer a montagem para obter a linguagem de máquina. No início, este procedimento é de forma manual e posteriormente através de um programa montador. A montagem manual esclarece alguns conceitos, relacionados à geração de código executável, escondidos nos ambientes de desenvolvimento integrados. Apesar da ferramenta simular um processador hipotético simplificado, ela foi construída seguindo a sintaxe de instruções usada na arquitetura Intel de 32 bits (IA-32), permitindo que o aluno utilize os conceitos absorvidos quase diretamente no entendimento de outras linguagens de montagem em computadores reais. Esta ferramenta é utilizada introdutoriamente na disciplina Linguagem de Montagem do Curso de Ciência da Computação na Universidade Estadual Paulista em Bauru com significativo aumento da taxa de aprovação dos alunos, a qual pode ser confirmada a partir dos dados apresentados sobre oito anos lecionados, sendo quatro deles com o uso do simulador.

PALAVRAS-CHAVE: Linguagem de Montagem. Simulador. Ensino de Computação.

NOTES

[1] GNUSIM8085 was written by Sridhar Ratnakumar in 2003. Available at: http://gnusim8085.srid.ca/. Access on: Dec. 10th, 2019.

[2] Processing. Available at: https://processing.org/. Access on: May. 15th, 2019.

REFERENCES

ACM/IEEE. Joint Task Force on Computing Curricula. **Association for Computing Machinery (ACM) and IEEE Computer Society:** Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science. ACM and IEEE Computer Society, 20 de dez. de 2013. Available at: https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf. Access on: Apr. 21st, 2020.

ACM/IEEE. Joint Task Force on Computing Curricula. **Association for Computing Machinery (ACM) and IEEE Computer Society:** Computer Engineering Curricula 2016: Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering. ACM and IEEE Computer Society, 15 de dez. de 2016. Available at: https://www.acm.org/binaries/content/assets/education/ce2016-final-report.pdf. Access on: Apr. 21st, 2020.

ESMERALDO, G.; LISBOA, E. B. Uma Ferramenta para Exploração do Ensino de Organização e Arquitetura de Computadores. **International Journal of Computer Architecture Education**, v. 6. p. 68-75, 2017.

ESMERALDO, G. *et al*. Um Estudo Comparativo entre Simuladores Computacionais para Apoio à Disciplina de Arquitetura e Organização de Computadores. In: CONGRESSO SOBRE TECNOLOGIAS NA EDUCAÇÃO (CTRL+E), 4., 2019, Recife. **Anais do IV Congresso sobre Tecnologias na Educação.** Porto Alegre: Sociedade Brasileira de Computação, p. 434-443, 2019.

ESMERALDO, G. Á. *et al*. CompSim: An Integrated Environment for Learning and Designing of Embedded Computational Systems. In: 2020 **XIV Technologies Applied to Electronics Teaching Conference (TAEE). IEEE**. p. 1-4, 2020.

INTEL. **Intel® 64 and IA-32 Architectures Software Developer's Manual.** Volume 1: Basic Architecture. p. 482, 2016**.** Available at: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-1-manual.html. Access on: Aug. 21st, 2018.

KOSTADINOV, N.; BENCHEVA, N. An Approach for Teaching Processor Design and How to Extend its Features. In: 2019. **29th Annual Conference of the European**

Association for Education in Electrical and Information Engineering (EAEEIE). IEEE, p. 1-4, 2019.

MALVINO, A.P. **Microcomputadores e Microprocessadores.** McGRAW-HILL do Brasil, 578 p., 1985.

MANSOUR, A. Y.; NAVIN, A. H.; RAHMANI, A. M. J1 Accumulator-Based Processor for Educational Purposes. **International Journal of Advanced Research in Computer Science**, v. 4 n. 4, p. 191-194, mar.-abr. 2013

MARIOTTI, G.; GIORGI, R. **WebRISC-V:** A RISC-V Educational Simulator featuring RV64IM, Pipeline and Web-Based UI. Department of Information Engineering and Mathematics, University of Siena, July 2020.

NIKOLIC, B. *et al*. A Survey and Evaluation of Simulators Suitable for Teaching Courses in Computer Architecture and Organization, **IEEE Transactions on Education**, v. 52, n. 4, p. 449-458, nov. 2009.

NULL, L.; LOBUR, J. MarieSim: The MARIE Computer Simulator. **ACM Journal on Educational Resources in Computing (JERIC)**, v. 3, n. 2, p. 1-29, 2003.

PATT, Y. N.; PATEL, S. **Introduction to Computing Systems:** From Bits and Gates to C and Beyond**.** 2. ed. New York, NY: McGraw-Hill Higher Education. 632 p., 2004.

SANTA, F. M.; SARMIENTO, F. H. M.; ARIZA, H. M. Minimalist 4-bit Processor Focused on Processors Theory Teaching. **Indian Journal of Science and Technology**, v. 10, n. 14, p. 1-6, 2017.

SARTOR, M.; SOARES, T. T. M. S.; BEREJUCK, M. D. Building a microprocessor architecture at Computer Engineering undergraduate courses. Building a microprocessor architecture at Computer Engineering undergraduate courses. **International Journal of Advanced Engineering Research and Science**, v. 7, n. 7. Jul. 2020.

SILVA, G. P.; BORGES, J. A. S. SimuS - Um Simulador Para o Ensino de Arquitetura de Computadores. **International Journal of Computer Architecture Education (IJCAE)** v. 5, n. 1, p. 7-12, 2016.

SILVA, G. P.; BORGES, J. A. S. A Didactic Processor and Simulator for IoT. In: 2018 3rd International Conference of the Portuguese Society for Engineering Education (CISPEE), Aveiro. 2018 **3rd International Conference of the Portuguese Society for Engineering Education (CISPEE)**, 2018. v. 1, p. 1-7, 2018.

VERONA, A. B.; MARTINI, J. A.; GONÇALVES, T. L. SIMAEAC: Um Simulador Acadêmico para Ensino de Arquitetura de Computadores. **Varia Scientia**, v. 9, n. 16, p. 139-148, 2009.

Received: Aug. 15th, 2020.
Approved: Nov. 23rd, 2021.
DOI: 10.3895/rbect.v15n2.13014
How to cite: PEGORARO, R.; FRANCHIN, M. N. A simulator to support assembly language teaching. **Brazilian journal of Science teaching and Technology**, Ponta Grossa, v.15, p. 1-19, 2022. Available at: <https://periodicos.utfpr.edu.br/rbect/article/view/13014>. Access on: XXX.
Mailing address: Rene Pegoraro - rene.pegoraro@unesp.br