An educational CPU Visual Simulator

Renato Cortinovis
Independent researcher
rmcortinovis@gmail.com

Abstract

This paper describes the redesign and extension of an old but effective educational CPU visual simulator. The main goal is to support novices in understanding the behaviour of the key components of a CPU, focusing on how code written in high-level languages is actually executed on the hardware of a computer. Extensions include the addition of CPU flags and related conditional jump instructions to better illustrate the control flow in low-level languages; the possibility to define and use labels for numerical addresses, in order to clarify the concept of variable as well as the mapping of high-level programming constructs to assembly language; enhanced color-coded animations to better understand the sequential nature of the control unit and the role of the control bus in addition to the address and data buses. While the old simulator was based on a Harvard architecture, the new one is based on the classical Von Neumann architecture, to illustrate in simpler terms the concept of stored-program computer. Additional enhancements include a new, more realistic, compare instruction (with two addressing modalities), bit-wise logical operations, and operational improvements such as simulation speed control and better code editing functionalities.

The new simulator has been developed following an Open Pedagogy / OER-enabled pedagogy approach, where a group of students incrementally modified the old simulator as part of their educational activities. This approach reduced the time spent on "disposable" traditional assignments, challenging students to address a real-world professional problem. Making available the result of these efforts with an open licence, we aspire to contribute to a self-fuelling cycle which will hopefully continuously improve and extend the resource for future students and teachers.

1. Introduction

It is well known that often students, despite studying both programming with high-level languages as well as the basics of computer architecture, do not fully grasp how code written in high-level languages is actually executed on the hardware of a computer (Evangelidis et al., 2001; Miura et al., 2004). The main goal underlining this activity, was to develop a tool to support this understanding.

A wide range of simulators capable to visualize the execution of the low-level operations in a computer, notably including CPUs, have been developed with this aim. Decker and Hirshfield (1998) developed a simplified CPU visual simulator, called PIPPIN, which supported an essential set of instructions, whose execution was dynamically visualized in a simplified functional architecture (Figure 1).

The CPU simulator was complemented with another tool, Rosetta, to translate arithmetic expressions of assignment statements directly to PIPPIN code. The main focus of these combined tools was indeed the parsing and translation of arithmetic expressions to assembly code, and its visual execution. PIPPIN and Rosetta were developed as Applets, certainly a suitable technology at that time, and were associated with the book: "The Analytical Engine: An Introduction to Computer Science Using the Internet" (Decker and Hirshfield, 2001). The goal of the authors was not to develop very comprehensive and universal tools, but rather simplified tools to show students "enough about how computers operate to convince them that the rest is mere details" (Decker and Hirshfield, 1998). The tools were quite popular: the book sold thousands of copies (S. Hirshfield, personal communication, April 2, 2021), and there were versions openly available on the Internet.

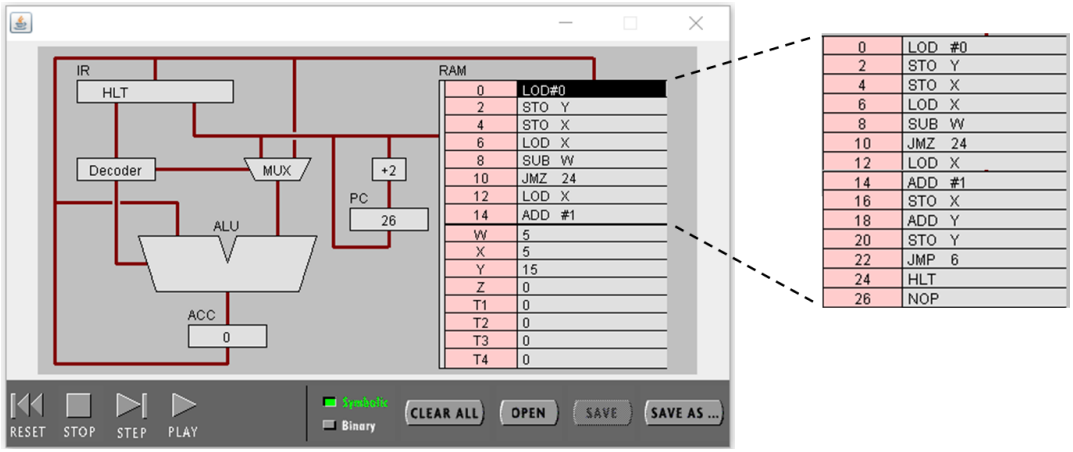

Figure 1 – A screenshot of the original PIPPIN (with an expanded section on the right)

The author of this paper too made use of PIPPIN for almost two decades in adult-education specialized computer-science courses in Italy (ITIS P. Paleocapa – Bergamo), frequently combined with other tools such as the Little Man Computer or emulators of real processors, attempting to avoid their weaknesses while exploiting their advantages. Yet, as will be discussed, mixing different tools did not prove to be the best solution. This multi-year experience teaching and observing students experimenting with the simulator, including discovering the difficulties they faced as well as the misconceptions and bad habits that sometimes they picked up, helped to identify desirable extensions. Notably, the porting of the tool (from an obsolete applet to a Java application) and the implementation of the extensions were carried out by the students themselves, in the context of open educational practices as part of their educational activities in object oriented programming in Java. The paper is organized as follows: Section 2 identifies and motivates the requirements, Section 3 illustrates the new design, Section 4 presents the development process as an open-pedagogy activity, and Section 5 concludes by discussing future plans.

2. Requirements and their rationale

As previously introduced, the attempted solution to support students in understanding how high-level code ends up being executed in a computer, has been for many years to mix different tools, trying to cherry pick their best features. PIPPIN is very simple and supports effective visual animations, but lacks some important features. For example, as noted by Miura et al. (2004, p. 1067) too, it focuses on the translation of assignment expressions but "it is insufficient for students to understand the whole process of program translation followed by execution in a computer". The Little Man computer, conceived by Madnick in the sixties at MIT, is a widely known instructional model to explain the basic architecture and operations of computers. Many similar visual implementations exist (Yurcik and Osborne, 2001), and I used in particular the excellent implementation by Higginson (2014). The Little Man Computer simulator has a more comprehensive instruction set than PIPPIN, and good animations too, but it does not represent with sufficient realism the functional architecture of a computer. For example, the memory is represented as a set of mailboxes, there is no representation of the CPU internal paths, and it is not possible to switch to a binary visualization of the information (code and data). The emulators are obviously realistic by definition, but their complexity can be overwhelming for less experienced students, and they lack detailed animations desirable for novices. Finally, such a variety of tools use different instruction sets, slightly different architectures (for example linear memory versus segmented memory), and different operational environments. Hence, the students waste too much time familiarizing themselves with each tool and get confused by the different details: many weak students end by giving up.

In synthesis, the strategy to integrate different tools did not prove to be an effective solution: it would be far better to have a single tool, as simple as possible but at the same time comprehensive enough to support the explanation and experimentation of a few more aspects than what was possible with PIPPIN. The following subsections discuss the most relevant aspects. Additional ones were addressed, but not discussed here in detail, such as:

- The need to differentiate more clearly the two subsystems CPU and RAM, including differentiating with colour coding the internal versus external buses, because it was not always evident to students what are the boundaries between CPU and RAM in PIPPIN (Figure 1);
- Improvements in the usability of the interface:
    - the possibility to insert new instructions in the middle of existing code without having to rewrite the portion of code below the insertion,
    - the need to let users control the speed and the amount of details in the animations, because PIPPIN is very slow and cumbersome to use when students work with algorithms including lengthy cycles.

2.1. Status word with flags and orthogonal control flow instructions

PIPPIN does not foresee a status word comprising a set of flags to indicate conditions such as zero, negative, even, carry, and so on, which are usually automatically set by the execution of any arithmetic or logic operations. In order to control the flow of execution, the PIPPIN instruction set includes an unconditional jump instruction, and just a single conditional jump instruction which jumps if the accumulator is zero. Supporting instructions include "compare zero" and "compare less" instructions to set the accumulator to 1 when their parameter is, respectively, equal or less than zero, and a "not" instruction to toggle the accumulator between 1 (actually anything different from 0) and 0. While this choice was probably originated by the desire to keep the simulator as simple as possible, avoiding the need to introduce the concept of status word / flags, students find it very cumbersome. They find it easier to work with real processor emulators, having a status word and an orthogonal set of conditional jump instructions. There, the flags are automatically updated to comply with the results of every arithmetic or logic operation, and can be exploited by corresponding instructions such as jump on zero, jump on not zero, jump on negative, jump on not negative, etc. It was frequently observed that the PIPPIN solution, only apparently easier, leads students to go astray with convoluted ad-hoc solutions: it does not support students to figure out systematic translation patterns from high level constructs (IF-THEN-ELSE, WHILE-DO, etc.) to assembler instructions, as shown in Figure 2.

```
translation (
        instruction1
        WHILE <condition> DO
                instruction2
                instruction3
        ENDWHILE
        instruction4
)
```

```
                translation (instruction1)
WHILE:          translation(<condition>)
                JMP !condition, ENDWHILE
                translation (instruction2)
                translation (instruction3)
                JMP WHILE
ENDWHILE: translation (instruction4)
```

Figure 2 – Sample translation pattern of a WHILE-DO construct

Hence, a fundamental new requirement was to support a status word automatically updated by every arithmetic or logic operation, and a set of corresponding orthogonal conditional jump instructions.

2.2. Labels: proper naming of variables and symbolic identifiers for instruction addresses

PIPPIN has only a few pre-specified labels that can be used as variable names: W, X, Y, Z, T1-T4. This way students, not being permitted to choose meaningful names in line with their intended use (semantics) such as SUM, COUNTER, or INDEX, are incentivized to pick up the bad habit of using meaningless names for variables. I even noticed students who, after working with PIPPIN, developed code in Java using T1-T4 as variable names. Furthermore, the inability to define new labels does not help students to grasp the concept that basic "variables" are essentially aliases for memory addresses. This is further aggravated by having variables and instructions allocated in separate memory segments, implementing a Harvard architecture, rather than the conceptually simpler Von Neumann architecture.

The possibility to associate labels to memory addresses, makes it possible to use meaningful identifiers for variables, plus it can be conveniently used to specify symbolic parameters in control flow

instructions too, such as “jump ENDWHILE”, where ENDWHILE is a label indicating the memory address where the next instruction to be executed is stored (Figure 2). This greatly facilitates students in better mapping high-level code to low-level code. The possibility to dynamically switch from symbolic visualization to binary, greatly helps students to understand that labels are just aliases for numerical addresses.

The new requirement was, therefore, to let users define their own labels as aliases of memory addresses, to be used both as meaningful variable names as well as parameters in jump instructions. Additionally, it was considered convenient to adopt a conceptually simpler Von Neumann architecture rather than a Harvard one.

2.3. Misunderstandings about the sequential interaction between CPU and RAM

PIPPIN depicts a “Decoder” box, which selects the input channel to the ALU via a multiplexer, and determines the specific operation performed by the ALU, according to the content of the instruction register as its input (Figure 1). This apparently reasonable simplification creates some confusion among the students, who are left wondering how a pure combinatorial network (the decoder) can handle all the controls necessary to the system. This made it necessary to provide at least some visual clues that there is indeed a subsystem (the control unit) managing these aspects of a sequential nature.

In particular, the most inquisitive students typically ask how the RAM can figure out that the data have arrived, when they consist of all zeros. These students are obviously led to incorrectly believe that the RAM knows that there are new data as soon as the data lines are not zero. Students also frequently ask how the RAM knows whether it needs to store or retrieve data. Indeed, PIPPIN does show, and properly animates, the data and address buses both internally to the CPU and externally towards the RAM, but no control line is shown between the CPU and RAM.

To help eliminate these drawbacks, it was considered necessary to explicitly indicate the (sequential) control unit, which includes the decoder, and explicitly show and properly animate the information transmitted via the control bus between the CPU and memory. The complexity of this configuration, also calls for a more clear differentiation of data, addresses and control buses with colour coding techniques.

3. The new design

Figure 3 shows a screenshot of the new Educational CPU Visual Simulator, with a sample program computing the sum of all numbers from 1 to MAX (which has value 5 in this case), loaded in RAM.

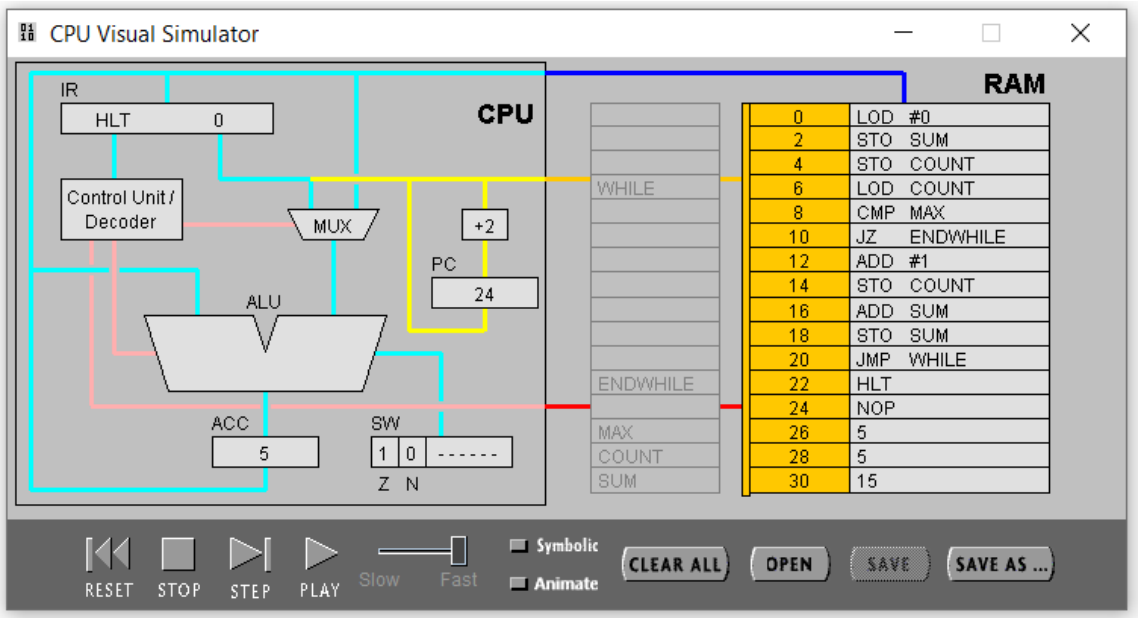

Figure 3 – A screenshot of the new CPU Visual Simulator

The image exemplifies how the new design satisfies the requirements discussed. The system now models a conceptually simpler Von Neumann architecture, rather than the previous Harvard one: instructions and data are therefore stored in the same memory segment. CPU and RAM are now more clearly separated and labelled. There is a new Status Word register with "zero" (Z) and "negative" (N) flags, which can be conveniently used with the corresponding conditional jump instructions: jump on zero, jump on not zero, jump on negative, and jump on not negative. The "Decoder" has been replaced by a "Control Unit / Decoder" box, from where additional control lines reach the RAM. This control bus is now colour coded and animated, showing the Read or Write requests to the memory with appropriate timings. The data and address bus too are now colour coded for easier understanding, and obviously animated as they were in PIPPIN. Internal and external buses are also differentiated with colour coding.

There is an extra virtual table where the user can easily define arbitrary labels corresponding to physical memory addresses, which can then be used as variable names or as parameters of jump instructions. In the sample listing visible in the memory in Figure 3, it is possible to appreciate the use of semantically meaningful names for the variables (COUNT, SUM, MAX), as well as the evident pattern of a high-level WHILE-DO control structure, corresponding to the high-level pseudocode in Figure 4. The advantage of this solution can be better appreciated by contrasting Figure 3 with Figure 1, showing the coding of the same algorithm in the old simulator.

```
// sum all the integer numbers from 0 to MAX
// assertion: MAX >= 0

SUM = 0
COUNT = 0
WHILE COUNT != MAX DO
   COUNT = COUNT + 1
   SUM = SUM + COUNT
ENDWHILE
```

Figure 4 – pseudocode corresponding to the assembler code in Figure 3

The compare instruction (CMP with both immediate and direct addressing) visible in the code in Figure 3, integrates and extends the previously existing compare zero (CMZ) and compare less (CML) instructions. It works exactly like a SUB (subtract) instruction, updating the zero and negative flags, but it does not modify the content of the accumulator. Again, the advantage of these extensions can be appreciated by contrasting the listings in Figure 3 and Figure 1. Other interventions not visible from Figure 3, include replacing the logical operations (AND and NOT) with more commonplace bitwise logical operators.

In addition to the possibility to switch between symbolic and binary visualizations, a few additional controls allow the user to dynamically modify the speed of the detailed animations, or even switch them off once the basic mechanisms have been mastered and students move to more complex and time consuming code.

4. Open-pedagogy approach

The new simulator was developed following an Open Pedagogy / OER-enabled pedagogy approach (Wiley and Hilton, 2018), where a group of students modified the old simulator, carrying out all the necessary design, refactoring and coding, as part of their educational activities in a programming project. The author first discussed with them the requirements, then – following an iterative, incremental fast prototyping approach – provided them prompt (asynchronous and synchronous) feedback on their requests for clarifications and on their prototypes. The objective was to make the result of their effort openly available for use and for further improvements by other students and teachers. This approach challenged students to address a real-world professional problem, and avoided the need to waste

resources on "disposable" traditional assignments. The feedback from the students involved was very positive in general:

"Absolutely a very satisfying educational experience".

More specifically, they appreciated the positive impact on their motivation:

*"I was impressed to experience how solving real problems such as introducing a new feature in a broader complex application, had such a positive impact on my motivation and my **passion** for software development",*

as well as the opportunity to work in team and to learn from professionally-written code:

"Working on a team project rather than developing 'write-and-forget' software allowed us students to simulate a working environment where we had the chance to learn from professionally-written code and apply changes that we, as previous users, think would be beneficial to future students."

Other students who did not have yet the necessary competences to intervene in the code of the simulator, were anyway involved in the project, testing the application and producing supporting documentation. The project was also taken as an opportunity to discuss in general the open software movement (Carillo and Okoli, 2008), and the potential contribution of open educational resources to the sustainable development goals (Lane, 2017). All the students were thrilled by the opportunity to give their contribution for the common good. One of them, for example, commented:

"Working for a purpose and contributing to a wider goal is definitely more rewarding than getting a good score on a typical classroom assignment for its own sake"

These human values, responsibilities and soft skills, are increasingly considered a fundamental complement to sheer technical competences, and deserve to be deliberately targeted in the education of software engineers (Goyal and Capretz, 2021).

Finally, it is worthwhile mentioning that all the students involved in the development of the software, could find a job in the Information Technology sector even before completing their studies. While a precise relationship of cause/effect cannot be claimed, there is some evidence that such an activity was an opportunity for these bright minds to blossom.

5. Conclusions and future activities

The previous PIPPIN was an excellent educational CPU visual simulator from many points of view, but focusing on the translation of arithmetic expressions, lacked a few other functionalities considered fundamental by the author. The new simulator builds on the strengths of the previous one, but it supports a more direct mapping from high level control structures to assembler patterns, it improves the understanding and proper naming of variables, it is more realistic without being more complex, and it provides a better user experience. It is now a very flexible tool that can potentially be used by teachers and learners in many different ways: to show how a computer works, to understand the key components of a CPU and see them alive during execution, to map high-level control structures to assembler patterns, to develop and visually run actual programs with a simple but representative language of 16 assembler instructions, etc.

Making available the result of these efforts with an open licence, we aspire to contribute to a self-fuelling cycle which will hopefully continuously improve and extend the resource for future students and teachers. For example, the current simulator does not support the explanation and experimentation with another important aspect of programming: how to deal with subprograms, including parameters passing and local variables. Hence, a desirable additional extension would consist in adding a Stack Pointer register with related push and pop operations, as well as instructions to call, and return from, subprograms. In order to satisfy the requirement of keeping the simulator as simple as possible, this added layer of functionalities could be only optionally exposed under user control, keeping the simulator as it is now by default. To further improve its usability, the simulator could be redeveloped from scratch, using JavaScript or any language that would allow the simulator to run directly in a browser: this would eliminate the current need to download the software and the Java runtime.

Certainly, given the very positive experience with the adopted OER-enabled pedagogy approach, the author will re-target its future educational activities aiming, whenever possible, to improve existing open educational resources, or to develop new ones, at the expense of less motivating "disposable" assignments. These activities will be designed to engage students with diverse competences, backgrounds, and inclinations. For example, while some students will write advanced software, others will develop supporting documentation and test it in peer-to-peer activities, or will localize the material to different languages and cultural contexts.

6. Acknowledgements

I would like to thank Paul Mulholland for his generous support in writing this paper, Stuart Hirshfield for his comments as one of the authors of the original tool, and my students who keenly engaged in this project, among them: Nicola Preda, Jonathan Cancelli, Alessandro Belotti, Davide Riva, Giordano Cortinovis, Giovanni Ingargiola, Mariapia Cavarretta, Alessandro Suru, and Piero Sileo. Last but not least, I would like to thank Cengage, in particular Kevin Kuhnell, for generously granting us the permission to modify, reuse, and republish the original applet (for non-commercial, educational purposes only).

7. References

Carillo, K., & Okoli, C. (2008). The open source movement: a revolution in software development. *Journal of Computer Information Systems*, 49(2), 1-9.

Decker, R., & Hirshfield, S. (1998). *The Analytical Engine: An Introduction to Computer Science Using the Internet*. PWS Publishing, Boston.

Decker, R., & Hirshfield, S. (2001). The PIPPIN machine: simulations of language processing. *ACM Journal of Educational Resources in Computing*, 1(4).

Evangelidis, G., Dagdilelis, V., Satratzemi, M., & Efopoulos, V. (2001). *X-compiler: Yet another integrated novice programming environment* [Paper presentation]. Proceedings IEEE International Conference on Advanced Learning Technologies, pp. 166-169. IEEE.

Goyal, D., & Capretz, L. F. (2021). *Promoting and Teaching Responsible Leadership in Software Engineering* [Paper presentation]. Proceedings of the 32nd Annual Workshop of the Psychology of Programming Interest Group (PPIG).

Higginson, P. (2014). Little Man Computer [Javascript application]. Retrieved 9/4/2021 from: https://peterhigginson.co.uk/LMC/.

Lane, A. (2017). Open Education and the Sustainable Development Goals: Making Change Happen, *Journal of Learning for Development - JL4D*, 4(3), pp. 275-286.

Miura, Y., Kaneko, K., & Nakagawa, M. (2004). *Development of an educational computer system simulator equipped with a compilation browser* [Paper presentation]. Proc. Int'l Conf. Computers in Education, pp. 1067-1071.

Wiley, D., & Hilton, J. (2018). Defining OER-enabled Pedagogy. *International Review of Research in Open and Distance Learning*, 19(4). DOI: 10.19173/irrodl.v19i4.3601.

Yurcik, W., & Osborne, H. (2001, December). *A crowd of Little Man Computers: Visual computer simulator teaching tools* [Paper presentation]. Proceedings of the 2001 Winter Simulation Conference, Vol. 2, pp. 1632-1639. IEEE.