

Frances-A: A Tool For Easy Realistic Architecture Level Program Visualization

Tyler Sondag, Kian L. Pokorny, and Hridesh Rajan

TR #10-08

Initial Submission: September 6, 2010.

Keywords: Frances, Visualization, Architecture

CR Categories:

K.3.0[Computers and Education] General K.3.2[Computers and Education] Computer and Information Science Education - computer science education, curriculum C.0[Computer Systems Organization] General – Modeling of computer architecture

Submitted.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Frances-A: A Tool For Easy Realistic Architecture Level Program Visualization

Tyler Sondag

Dept. of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA 50014
sondag@iastate.edu

Kian L. Pokorny

Division of Computing
McKendree University
701 College Road
Lebanon, IL 62254
klpokorny@mckendree.edu

Hridesh Rajan

Dept. of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA 50014
hridesh@iastate.edu

Abstract

Computer organization and architecture courses are an integral part of computer science education. For such courses, there are advantages to studying real architectures instead of simplified examples. However, real architectures are complex and difficult to grasp in a single semester course. Visualization of a running program on a given architecture can help with this problem by showing students functionality of architectural components and their interactive behavior. However, existing architectural simulations are difficult to use and consequently difficult to adopt in a course where time is already limited. To solve this problem, we present Frances-A. Key to Frances-A's design is that it uses a simple web interface that requires no setup and is easy to use, making it easy to adopt in a course. Frances-A also includes several features that further enhance its usefulness in a classroom setting. These features include graphical relationships between high-level code and machine code, clearly illustrated step by step machine state transitions, color coding to make instruction behavior clear, and illustration of pointers.

Categories and Subject Descriptors K.3.0 [Computers and Education]: General; C.0 [Computer Systems Organization]: General—Modeling of computer architecture

General Terms Education, Languages, Architecture

Keywords Frances, Visualization, Architecture

1. Introduction

Computer organization and architecture courses are a crucial component in computer science education [1]. The study of

a computer architecture and its behavior is a typical component to such a course. Often courses materials revolve around “toy” architectures. There are several advantages to learning a “real” architecture [29]. Learning a real architecture is difficult in a semester due to complexity. Since this is difficult, it is desirable to use interactive tools which have been shown to be highly effective for education [11, 29].

Many tools exist for simulating programs on different architectures [3, 4, 10, 16, 17, 19], however, they are typically time consuming to learn and difficult to use. As a result, adopting them in a course is challenging. At the same time, other difficult topics like machine language must be learned which may be significantly different from previous languages students have encountered.

To solve these problems, we present Frances-A, a tool for visualizing program execution and a realistic architecture. Frances-A includes several interesting features which enhances its usefulness in education. First, it provides a simple, easy to learn and easy to use web interface that requires no setup. Second, it graphically shows how familiar high-level code maps to machine code thus not requiring thorough knowledge of a machine language to start using the tool. Third, it shows graphically how each machine instruction impacts the machine state. Fourth, it allows both forwards and backwards stepping through program steps to allow students to revisit complicated steps and processes. Fifth, it color codes parts of the machine state to make the impacts of each instruction clear. Finally, difficult concepts surrounding addresses (e.g. pointers and stack) are clearly illustrated using color coded arrows.

By providing our simple web-based interface we avoid the three of the four biggest factors hindering adoption of such visualization tools in educational settings¹ namely time to learn the new tool, time to develop visualizations, and lack of effective development tools (we also eliminate other large problems such as reliability and install issues) [29]. Our

¹ We solve the fourth problem by making interesting examples (as lessons) available on Frances-A's website.

interface also displays the system state in a logical way and illustrates several important concepts. Further, backwards stepping is a rare feature that our tool includes. All of this together makes our tool easy to adopt, use, and understand.

The rest of this paper is organized as follows. Section 2 presents related ideas. Next, Section 3 describes our goals when developing Frances-A. Section 4 describes design and implementation of Frances-A. Finally, Section 5 discusses future work and concludes.

2. Related Work

2.1 Architectural Simulators

A large body of work exists for simulating architectures and teaching computer organization and architecture [2–4, 10, 13, 16, 17, 19, 23, 31, 34]. Many of these simulators are targeted toward advanced users. As a result they are typically very complex and difficult to learn. We now briefly discuss work in this area most similar to our introductory computer architecture pedagogical tool.

Null and Lobur developed MarieSIM [17] for use in teaching computer architecture and assembly language. This approach has several advantages including a simple assembly language and an accompanying text book. However, some argue that there are advantages to using “real” assembly languages rather than “toy” languages. This is a fundamental difference in our approaches. For those who prefer to use a “real” assembly language we recommend Frances-A. Further, our work differs in several ways. First, with MarieSIM there is a disconnect between high-level languages and the MARIE instruction set since users must program simulations with the 16 one address instructions provided with the system. With Frances-A, students have the option of entering simulations using a variety of high-level languages (or assembly). Thus, students can see how the system processes code they normally write and the learning curve for initial use of the tool is very low. Recall that MarieSIM has an accompanying text book. This has benefits, however, instructors that have designed their courses around different text books may not want to re-design their course around a new book. To help ease the adoption of Frances-A into existing courses, we develop our course materials around topics that compliment existing courses utilizing a standard textbook. Further, Frances-A is released via the web rather than requiring installation which hinders adoption due compatibility and dependence issues. Finally, MarieSIM does not allow “stepping” backwards through program execution. Frances-A has this feature to allow students to revisit previous execution steps without re-running the entire simulation.

Graham developed “The Simple Computer” simulator [10]. Stone later used this simulator to teach CS1 topics [26]. Another simulator developed by Braught and Reed is targeted toward introductory students in CS0 [4]. The main differences between these works is that (a) Frances-A

has a graphical interface that we believe has a much lower adoption time, and (b) Frances-A uses real instruction set architectures (ISA) rather than toy machines and languages.

Borunda *et al.* developed GSPIN, a MIPS simulator [3] for use in introductory computer architecture courses. This tool shows simultaneous views of the program call graph, intra-procedural control flow graph, MIPS assembly code, registers, etc. Our work differs in that Frances-A is not restricted to any one target language or architecture (though we leave simulation extensions for future work). Further, Frances-A more easily integrates with both high-level and low-level languages giving students the ability to visualize high-level code at a lower level and ease the learning curve. Finally, our control flow graph representations maintain the instruction ordering and layout of actual assembly programs. Thus, we believe Frances-A will be more effective when learning assembly language.

2.2 Program Visualization

A large body of work exists for software visualization [6, 7, 11, 14, 15, 18, 20–22, 27, 29, 30, 32, 33]. Price *et al.* developed a taxonomy for software visualization [18]. In this taxonomy, they make the distinction between algorithm visualization (illustrating high-level abstract code) and program visualization (illustrating actual code listings). First, rather than illustrating abstract algorithms, we focus on illustrating issues such as program implementation, language construct behavior, program state, and machine state. Therefore, we consider algorithm visualization to be complementary to our work on Frances-A. In terms of program visualization, a major difference between our work and previous work is that we believe our interface is much simpler than related projects. Thus, we address the primary factor limiting adoption of previous work [29]. We are able to do this because Frances-A’s backend consists of several powerful program analysis techniques. Rather than requiring installation, Frances-A is deployed via a web interface thus removing additional hurdles such as software and OS dependencies. Therefore, we avoid many of the factors which hurt adoption of such tools [29]. Finally, we are developing Frances-A using a realistic machine model and instruction set rather than toy models thus avoiding a disconnect between the tool and “real” language [29].

An example of a program visualization technique is a debugger such as gdb [9], or a graphical debugger such as kdbg [24], etc. Debuggers are very powerful and expressive tools and as a result are generally difficult to learn to use. Debuggers have several problems making them uncompatible with program visualization for pedagogical purposes. Their interface is typically highly expressive and thus overwhelms introductory students with details. Furthermore their interfaces do not visualize program structure. Finally, several aspects of the process may be confusing for introductory level students such as breakpoints, different techniques for stepping through execution, modifying program inputs, etc.

Our interface is only as expressive as necessary for introductory students, avoiding many complex features of debuggers. Finally our interface has fixed abstractions that allow us to eliminate issues like breakpoints and different techniques for stepping through execution.

Most similar to the proposed work is the work by Sundararaman and Back [27] on HDPV, a runtime state visualization tool for C/C++ and Java programs. This work is complementary to our own in two ways. First, the focus of HDPV is on visualization of data structures, whereas our focus is on control flow, system state, and program behavior. Second, they deal with representation concerns for large programs. Since our visualization is more geared toward introductory courses and not advanced courses or software engineering practice, we leave these large scale concerns for future work. HDPV uses a realistic machine model that captures low level details like memory layout. Since the focus is data structures, they do not trace register values. Register values do not appear until moved to memory. For introductory students this can be quite confusing. For example, loop counters often never go beyond registers. Since we focus on classroom use, we address this limitation by modeling registers as well as the stack and heap separately. Rather than focusing on specific languages, our work currently supports any language which can be compiled to native code. To enable several low-level implementations, HDPV makes use of Pin [5]. Similarly, we make use of cross-compilers and GNU BinUtils [8] to support a variety of targets. Frances-A allows the user to step backward in the code. HDPV does not.

Also similar is IBM's Jinsight tool [32]. The most important difference from our work is that Jinsight focuses on more advanced users and program behavior in terms of performance. In terms of implementation, Jinsight uses execution traces to develop its visualizations. Our tool uses an online based approach. A tool similar to Jinsight was also developed by Reiss [20]. This tool, like Jinsight, is targeted towards more experienced programmers and looks at phase behavior [28] and performance issues.

3. Goals for Frances-A

We had several goals in mind while designing Frances-A. Chief among these goals is an easy to use tool for introductory students learning low-level computer concepts. This includes being easy to learn, requiring no setup, and not requiring thorough knowledge of a machine language. Next, the tool needed to be effective. To ensure this, we desired a tool that would allow students to clearly visualize the behavior of each machine instruction.

3.1 Easy to use

We took several steps to make the tool as easy to use as possible. First, we desired a tool that required no setup thus avoiding issues such as software dependencies. This helps avoid adoptions hurdles regarding reliability issues and eases

tool investigation by potential educators. Next, we needed to design a simple interface that has very little learning curve. This is necessary to ensure that tools are feasible to adopt in a single semester without distracting students. To facilitate this simple interface, we rely on graphical features to show complex properties such as pointers, changes in state, accesses to memory locations, etc. This includes a simple logical layout of the system state. Another highly important goal was that we did not want to require students to have a thorough knowledge of machine language to use the tool. Since architecture courses are often the first exposure a student has to machine language, we wanted to ease this process as much as possible.

3.2 Effective

One of the main goals of our system to set it apart from others is ease of use, however, effectiveness is still critical. Without it, ease of use is useless. It has been shown that the way students interact with visualization tools is more important than the visualization itself [11]. Thus, we wanted a hands-on tool to improve the learning process. To make this hands-on tool effective, we had several goals.

1. *Support visualization on a real architecture and instruction set.* This would make the knowledge gained by using tool applicable to standard learning materials and in the real world.
2. *Allow students to enter simulation code in a familiar high-level language.* This would help students quickly visualize how the machine will handle familiar source code. This also allows students to more rapidly perform their experimentation instead of coding visualization code in an unfamiliar machine language, further decreasing learning curve.
3. *Allow students to step both forward and backward during program visualization.* This is a feature which is rare amongst such visualization tools. We believe this feature is crucial to allow students to revisit complex instructions and sequences of instructions without re-running the entire simulation that has the drawback that the student may lose context.
4. *Provide a graphical and logically organized layout.* We desired a graphical layout that was not only logically organized but color coded to show accesses and modifications to the machine state as well as concepts such as pointers and stacks.

4. Frances-A

We now describe the use and major features of Frances-A. To start using the tool readers may point their WWW browser to the URL <http://www.cs.iastate.edu/~sapha/frances-a>.

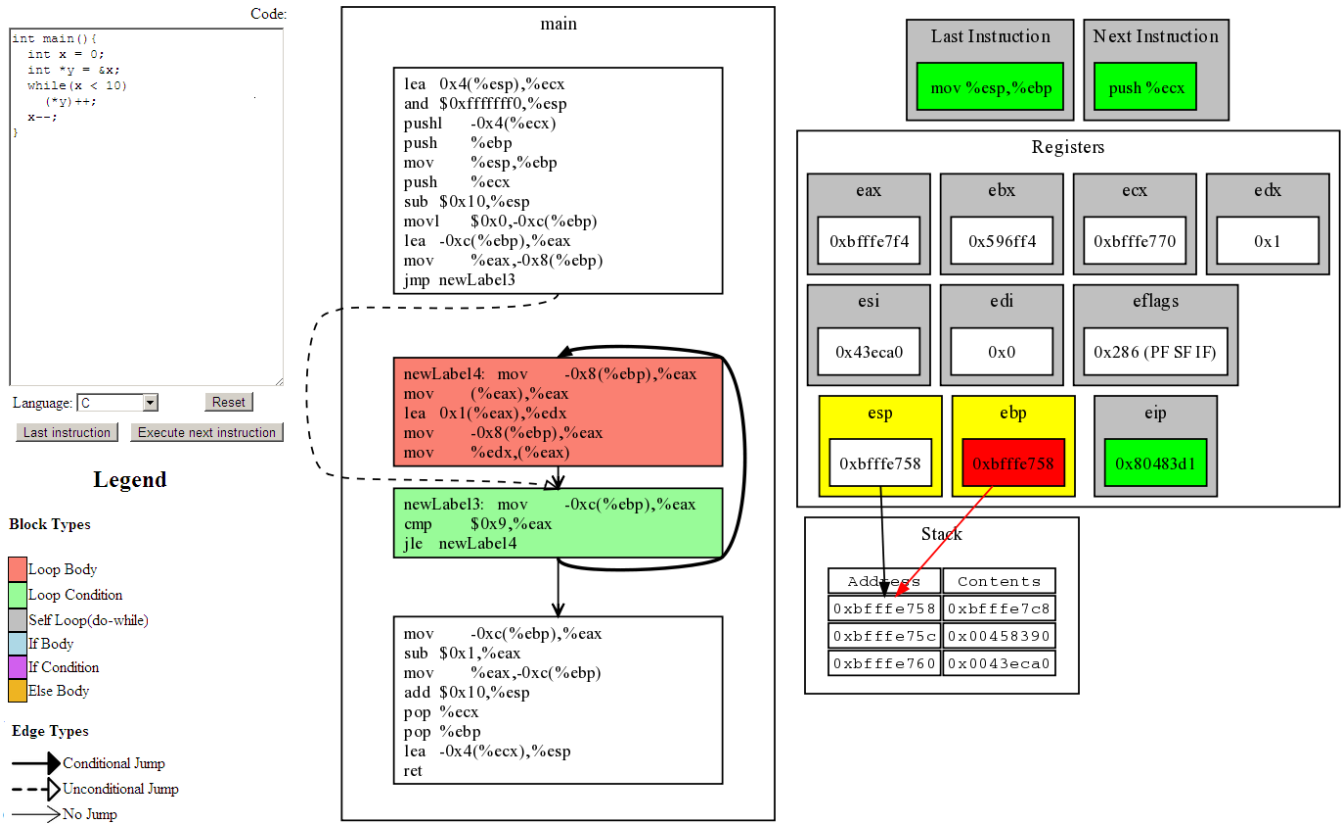


Figure 1. Simple while loop running through Frances-A. The left side shows the code entry, the middle shows the graphical representation of the machine code, and the right side shows state after the fifth instruction.

4.1 Interface

Key to Frances-A is a simple easy to use web-interface that requires no setup. An example of this interface is shown in Figure 1. For now, let us ignore the detailed aspects of the figure and focus on its major components. There are three main components.

High-level code Entry First, on the far left, there is a high-level code input box. Currently, code may be written in C, C++, and FORTRAN. Support may easily be added for any language that can compile down to native machine code. Initially, this box is editable so that users may enter their simulation code. After editing code, the user clicks the “Compile” button. At this pointer, the code entry box becomes read-only and the “Compile” button is replaced with buttons for stepping backwards and forwards in the assembly code.

In Figure 1 the “Compile” button has already been pressed and several steps have been executed. At this point the simple while loop code is no longer modifiable unless the “Reset” button is pushed.

Low-level code In the middle we show the machine code. This representation comes from previous work on the Frances tool [25]. This component is discussed in more detail in Section 4.2.

Machine state Finally, the far right side shows the machine state. Currently, this portion of the tool supports the x86 architecture. This portion of the interface consists of several logically separated components. Each of these and their behavior as part of the visualization will be described in detail in Section 4.3.

4.2 High-level to machine language relation

Recall that in the middle of the interface we have a graphical representation of the machine code which comes from previous work on the Frances tool which is targeted at teaching code generation [25]. This part of the interface allows users to easily see how their high-level code maps to machine code by using graphical features such as color coding and different edge drawing techniques.

In Figure 1 the middle portion shows the simple while loop. The pastel red block represents the loop body and the pastel green block represents the loop condition. This helps illustrate key differences in layout of the structure between the two languages. Also, consider the edge leaving the loop body going to the loop condition. This edge is drawn in bold to denote that it is a branch taken edge. The legend in the bottom left describes edge types and color codes, allowing students to quickly and easily identify the code constructs.

The purpose of this portion of the interface is to ease the burden when learning machine language. Students may enter visualization code in familiar high-level language, then see how their code is represented in machine code and how that code modifies the system state. This means that students do not have to write simulation code in machine language which speeds up the overall educational process and helps ensure that users know the meaning of the code they are visualizing. Finally, it also helps the user visualize how different program structures behave at the machine level.

Currently this portion of the tool supports AT&T and Intel x86 syntax as well as MIPS. This component has several interesting aspects. Since this component is not the focus of this work, we refer to the original paper on the tool for a thorough discussion [25].

4.3 Graphical layout of machine state

We now discuss in detail each aspect of the graphical representation of the machine state.

4.3.1 Previous / next instruction

The first part of the interface consists of the blocks marked “Last Instruction” and “Next Instruction”. As the labels suggest, these denote the previously executed instruction that gave the current state and the next instruction to be executed. For example, in Figure 1, a `mov` instruction was just executed and `push %ecx` is next.

This allows the user to find the current location in program execution. Then, they can consider the changes caused by the last instruction. For example, the user can see that in the last instruction the value in `%esp` should have been placed into `%ebp`. By inspecting the current state, the user can see that these two registers currently contain the same value. Next, they can try to determine the effects of the next instruction before it executes. For example, the user could predict that the stack will grow by a location which will contain the value in `%ecx`.

It is interesting to note that the “Next Instruction” box contains the actual next instruction, not just the next sequential instruction. That is, if the “Last Instruction” was a conditional jump and the branch is set to be taken, the “Next Instruction” is the target of the branch not the fall-through case.

4.3.2 Registers

Next, the system’s registers are separated from the rest of the state. Within this group of registers there are logical separations. In Figure 2, notice that the first row of registers are the general purpose registers `%eax`, `%ebx`, `%ecx`, and `%edx` (even though all of the registers are general purpose, many are typically used for specific purposes). In the next row, the two registers `%esi` and `%edi` are placed together since these are typically used for storing addresses for memory reads and writes (again, the programmer need not follow this). Also in this row is the `eflags` register which contains

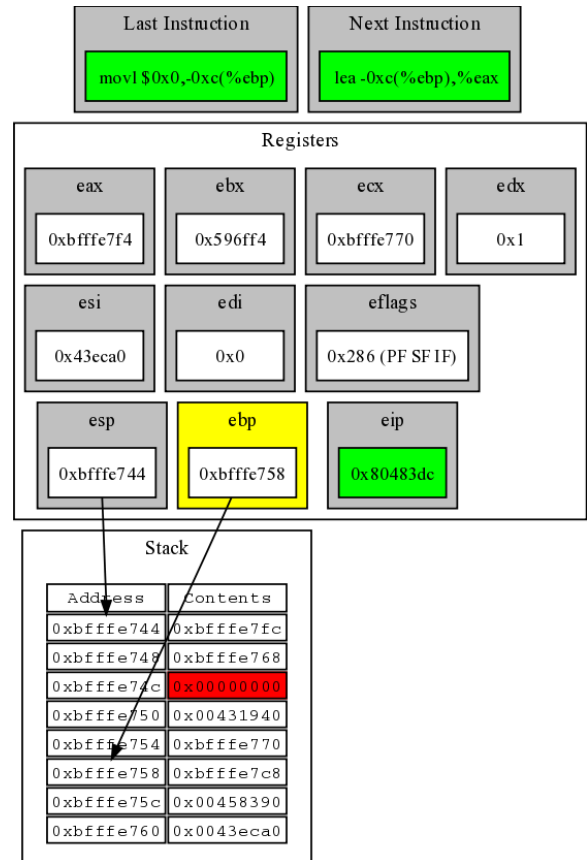


Figure 2. Frances-A state running the same code example as Figure 1 after the seventh instruction (`int x = 0;`).

the results of compare instructions as well as other secondary results of operations. In this figure, we can see that the PF, SF, and IF flags are set (all others are unset). The interested user is presented the the actual hex value of this register. In the final row, there are three pointer registers. The first two are stack pointers, `%esp` and `%ebp`. By looking at the values contained in the figure, one can determine that these addresses are located on the stack. The final register in this row is `%eip`, the instruction pointer.

4.3.3 Stack

Next, consider the representation of the stack. Figure 2, shows a stack containing 8 elements. Each element has its own row with columns specifying the address of that stack location and the contents at that location. The stack locations are important since they contain most local variables (some never leave the registers) as well as other temporary values. For example, in Figure 2, the last instruction moved the value 0 to third stack location. This corresponds to the assignment of `int x = 0;` from the code in Figure 1. The addresses are included in the representation so that users can see how contents of registers correspond to locations on the stack (e.g. stack pointers `%esp` and `%ebp`). In the figures, it is

easy to inspect the contents of these stack pointer registers and find the corresponding locations on the stack.

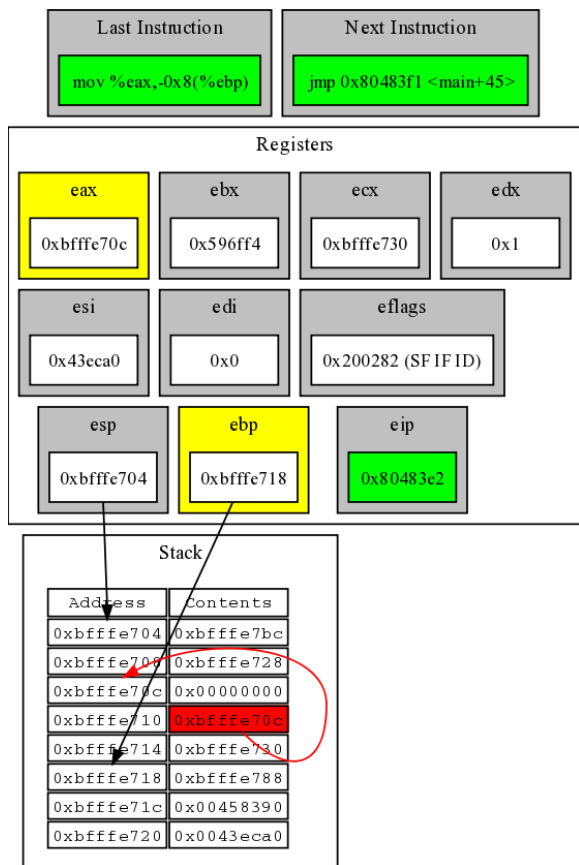


Figure 3. Frances-A state running the same code example as Figure 1 after the tenth instruction (`int *y = &x;`).

4.3.4 Edges

Now, let us consider the edges in the figures. First, in Figure 2 notice that there are two edges, one for both of the pointers into the stack. It is easy to see where these two registers point to on the stack. If the user is stepping through the simulation step by step, it is easy to see that the `%ebp` register points to the location on the stack before the 4 locations are added by the `sub $0x10, %esp` instruction. From the figure it is clear that the `%esp` register points to the top of the stack. This helps illustrate the purposes of the `%esp` and `%ebp` registers.

Also interesting to note are the edges in Figure 3. In this figure, the last instruction executed moved the value in `%eax` (address of `x`, `&x`) to the fourth location on the stack. As a result, there is now an edge from the stack location containing this value to the stack location corresponding to variable `x` (third location on the stack). This illustrates the behavior of pointers in the machine.

4.3.5 Color coding

At this point, we have ignored the color coding. We now describe this aspect of Frances-A which helps illustrate the purposes of the instructions and their impact on the machine state.

First, the green boxes are used to denote portions of the state that always change and are not referenced directly (previous and next instructions as well as instruction pointers). The purpose of separating out these components is to make it clear for users not to spend too much time trying to understand how these components are directly impacted by the instructions since they are only implicitly impacted. Note that the flags are not included in this scheme since they are not updated at each step.

Next, consider the color coding of the boxes for registers. Yellow boxes around the register contents signify that the last instruction accessed these registers (either read or write). For example, in Figure 3, we can see that the `%eax` register has been read (also note that it was the first operand in the last instruction). Additionally, the `%ebp` register has been accessed when calculating the stack location for the target of the operation.

Red highlighting of the register contents means that the contents was changed by the last instruction. Consider Figure 1 where the `%ebp` register has been modified to contain the value from `%esp`.

Similarly, we use red highlighting to show which stack contents have been modified. For example, in Figure 2, the value of 0 was assigned to the stack location corresponding to variable `x`. Thus, the corresponding stack location is red. This avoids the need for the user to try to determine the offset of the stack location manually.

Finally, consider the edge color coding. In Figure 3, notice that the two stack pointers are drawn in black whereas the pointer in the stack corresponding to variable `y` is drawn in red. Like the register and stack coloring, edges in red have been recently changed. In this example, since the pointer was modified the edge is red.

4.4 Reverse stepping

Another important feature of Frances-A which is rare among similar tools is the ability to step backwards through execution. We consider this a necessary feature that allows students to revisit complicated steps and groups of steps in the simulation. Note that reversing can also be simulated in a tool by rerunning the simulation, however, students may lose the context of simulation if too many steps are required for revisiting the previous instruction.

We color code changes to the state in red, however, this may not be enough. Thus, we allow students to go back so that they can revisit the state before it was modified. For example, when moving stack pointers, in order to understand behavior and purpose of the different pointers, it may be

important to go back to see the previous target of the pointer register.

Need for such a feature may be seen from Figure 1. We can see that the `%ebp` register has been changed to point to the top of the stack. However, we know nothing about where it pointed to previously. By simply clicking the “Last Instruction” button, we can clearly see the previous target of the register.

4.5 Backend

On the backend, we make use of both the original Frances tool [25] as well as the GDB debugger [9]. Aside from these, we use several scripts and programs to visually present the state as well as determine changes. After this, we use the GraphViz DOT program [12] to draw the visualization of the machine code and machine state.

5. Conclusion and Future Work

Knowledge of computer organization and architecture is a critical component in computer science education. To ease the process of learning real architectures and their behavior we introduce Frances-A. A key benefit of this tool is its usability in that it is easy to learn, easy to use, and requires no setup. Further, we have taken several steps to enhance its effectiveness. This includes logical separation of components of the machine state (including different register types), edge drawing to show pointer targets and stack behavior, color coding to show accesses and writes to show behavior of each instruction, and finally the ability to step both backwards and forwards through execution.

Future extensions to the tool involve the following. First, we may include more portions of the machine state (e.g. heap, floating point registers) to illustrate concepts such as dynamic allocation and data structures. For now, we focus less on data structures and more on visualizing simple programs. Second, we would like to add support for additional architectures such as MIPS (commonly used in education). Frances-A is designed to be easy to swap in different architectures, it simply requires installing additional software on the server and minor changes to the layout. Finally, we also plan to conduct a thorough evaluation of the tool (both through experts and students) to ensure both the usability and effectiveness of the tool.

Acknowledgements Sondag and Rajan were supported in part by US NSF under grants 06-27354, 07-09217, and 08-46059.

References

- [1] Computing curricula 2008: An interim revision of cs 2001. <http://www.acm.org/education/curricula/Computer-Science2008.pdf>.
- [2] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-oriented full-system simulation using M5. In *Workshop on Computer Architecture Evaluation using Commercial Workloads*, 2003.
- [3] Patrick Borunda, Chris Brewer, and Cesim Erten. GSPIM: graphical visualization tool for MIPS assembly programming and simulation. *SIGCSE Bull.*, 38(1):244–248, 2006.
- [4] Grant Braught and David Reed. The knob & switch computer: A computer architecture simulator for introductory computer science. *J. Educ. Resour. Comput.*, 1(4):31–45, 2001.
- [5] C.-K. Luk *et al.* Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.
- [6] Matthew J. Conway and Randy Pausch. Alice: easy to learn interactive 3D graphics. *SIGGRAPH Comput. Graph.*, 31(3):58–59, 1997.
- [7] Margarita Esponda-Arguero. Algorithmic animation in education—review of academic experience. *Journal of Educational Computing Research*, 39:1–15, 2008.
- [8] Free Software Foundation. GNU binutils: a collection of binary tools, May 2009. <http://www.gnu.org/software/binutils/>.
- [9] Free Software Foundation. GDB: The GNU Project Debugger, May 2010. <http://www.gnu.org/software/gdb/>.
- [10] Neill Graham. *Introduction to computer science (3rd ed.)*. West Publishing Co., St. Paul, MN, USA, 1985.
- [11] Christopher D. Hundhausen. Integrating algorithm visualization technology into an undergraduate algorithms course: ethnographic studies of a social constructivist approach. *Comput. Educ.*, 39(3):237–260, 2002.
- [12] J. Ellson *et al.* Graphviz - open source graph drawing tools. *Graph Drawing*, 2001.
- [13] K. Kise *et al.* The simcore/alpha functional simulator. In *WCAE '04: Proceedings of the workshop on Computer architecture education*, page 24, 2004.
- [14] K. Powers *et al.* Tools for teaching introductory programming: what works? *SIGCSE Bull.*, 38(1):560–561, 2006.
- [15] Myles McNally, Thomas L. Naps, David Furcy, Scott Grisson, and Christian Trefftz. Supporting the rapid development of pedagogically effective algorithm visualizations. *Journal of Computing Sciences in Colleges*, 23(1):80–90, 10/2007 2007.
- [16] Bosko Nikolic, Zaharije Radivojevic, Jovan Djordjevic, and Veljko Milutinovic. A survey and evaluation of simulators suitable for teaching courses in computer architecture and organization. *IEEE Transactions on Education*, 52:449 – 458, 11/2009.
- [17] Linda Null and Julia Lobur. MarieSim: The MARIE computer simulator. *J. Educ. Resour. Comput.*, 3(2):1, 2003.
- [18] Blaine A. Price, Ian S. Small, and Ronald M. Baecker. A taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4:211–266, 1992.
- [19] P.S. Coe *et al.* An integrated learning support environment for computer architecture. In *WCAE-3 '97: Workshop on Computer architecture education*, page 8, 1995.
- [20] Steven P. Reiss. Visualizing java in action. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, 2003.
- [21] Guido Rössling and J. Ángel Velázquez-Iturbide. Editorial: Program and algorithm visualization in education. *Trans. Comput. Educ.*, 9(2):1–6, 2009.
- [22] Dean Sanders and Brian Dorn. Jeroo: a tool for introducing

- object-oriented programming. In *SIGCSE*, 2003.
- [23] Herb Schwetman. Csim: a c-based process-oriented simulation language. In *Conference on Winter simulation*, 1986.
- [24] Johannes Sixt. A graphical debugger interface, May 2010. <http://www.kdbg.org/>.
- [25] Tyler Sondag, Kian L. Pokorny, and Hridesh Rajan. Frances: A tool for understanding code generation. In *SIGCSE*, 2010.
- [26] Jeffrey A. Stone. Using a machine language simulator to teach CS1 concepts. *SIGCSE Bull.*, 38(4):43–45, 2006.
- [27] Jaishankar Sundararaman and Godmar Back. HDPV: interactive, faithful, in-vivo runtime state visualization for C/C++ and Java. In *Symposium on Software visualization*, 2008.
- [28] T. Sherwood *et al.* Automatically characterizing large scale program behavior. In *ASPLOS*, 2002.
- [29] T.L. Naps *et al.* Exploring the role of visualization and engagement in computer science education. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 131–152, 2002.
- [30] Jaime Urquiza-Fuentes and J. Ángel Velázquez-Iturbide. A survey of successful evaluations of program visualization and algorithm animation systems. *Trans. Comput. Educ.*, 9(2):1–21, 06/2009 2009.
- [31] V. Pai *et al.* Rsim: Rice simulator for ilp multiprocessors. *SIGARCH Comput. Archit. News*, 25(5):1, 1997.
- [32] W.D. Pauw *et al.* Visualizing the execution of java programs. In *Revised Lectures on Software Visualization, International Seminar*, pages 151–162, 2002.
- [33] Ursula Wolz, John Maloney, and Sarah Monisha Pulimood. 'scratch' your way to introductory cs. In *SIGCSE*, 2008.
- [34] Hui Zeng, Matt Yourst, Kanad Ghose, and Dmitry Ponomarev. MPTLsim: a cycle-accurate, full-system simulator for x86-64 multicore architectures with coherent caches. *SIGARCH Comput. Archit. News*, 37(2), 2009.