FRANCES-A: A TOOL FOR ARCHITECTURE LEVEL PROGRAM VISUALIZATION

Tyler Sondag
Iowa State University
sondag@iastate.edu

Kian L. Pokorny
McKendree University
klpokorny@mckendree.edu

Hridesh Rajan
Iowa State University
hridesh@iastate.edu

ABSTRACT
Integral to computer science education are computer organization and architecture courses. We present Frances-A, an engaging investigative tool that provides an environment for studying real assembly languages and architectures. Frances-A includes several features that enhance its usefulness in the classroom such as graphical relationships between high-level code and machine code, illustrated step by step machine state transitions, color coding to make instruction behavior clear, and illustration of pointers. Frances-A uses a simple web interface requiring no setup and is easy to use, making it easy to adopt in a course.

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. Such components often revolve around "toy" architectures. Although there are advantages to learning a "real" architecture [17], it is complex and difficult to compress this into a semester; therefore, it is desirable to use interactive tools which have been shown to be highly effective for education [8, 17].

Many tools exist for simulating programs on different architectures [2–4, 7, 11, 13], 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 on a realistic architecture. Frances-A includes features that enhance its usefulness in education. First, it provides a simple, easy to learn and use web interface requiring 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 forwards and backwards stepping through the program allowing students to revisit complicated steps. Fifth, it color codes parts of the machine state to make the impact of each instruction clear. Finally, difficult concepts surrounding addresses (e.g. pointers and stack) are illustrated using color coded arrows.

By providing a simple web-based interface three of the four biggest factors hindering adoption of such visualization tools in educational settings[1] are avoided, namely time to learn the new tool, time to develop visualizations, and lack of effective development tools (other problems such as reliability and install issues are also eliminated) [17]. The interface 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 the Frances-A tool easy to adopt, use, and understand.

RELATED WORK
Architectural Simulators
A large body of work exists for simulating architectures and teaching computer organization and architecture [2–4, 7, 11, 13]. There are also many simulators targeted toward advanced users that 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 [11] for use in teaching computer architecture and assembly language. MarieSIM uses a simple assembly language and has an accompanying textbook. Further, MarieSIM requires users to program simulations in the machine language whereas Frances-A gives students the option

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

of entering simulations using high-level languages. Thus, the learning curve is low and students can visualize code they typically write. To help ease the adoption of Frances-A into existing courses, course materials have been developed around topics that compliment existing courses utilizing a standard textbook.

Other related projects include Graham's "The Simple Computer" [7] which uses a command line interface and custom ISA, and GSPIM [3], a MIPS simulator which also shows control flow graphs, but does not maintain actual instruction ordering.

Program Visualization

A large body of work exists for software visualization [5,8,10,12,14,16–20]. Price *et al.* [12] make the distinction between algorithm visualization (abstract code) and program visualization (actual code). We focus on program visualization and consider algorithm visualization to be complementary to Frances-A. A major difference between this work and previous work is that we believe the Frances-A interface is much simpler than related projects. This addresses the primary factor limiting adoption of previous work [17]. This is able to be done because Frances-A's backend consists of several powerful program analysis techniques. Rather than requiring installation, Frances-A is deployed via a web interface, removing hurdles such as software and OS dependencies. Finally, Frances-A is developed using a realistic machine model and instruction set rather than toy models, avoiding a disconnect between the tool and "real" language [17].

Examples of program visualization techniques include debuggers (e.g. gdb [6]) and graphical debuggers (e.g. kdbg). These tools are often difficult to learn due to their power and expressiveness. Furthermore they do not visualize program structure. Also, debugging concepts such as breakpoints, different techniques for stepping through execution, etc. may be confusing at first. Frances-A's interface is only as expressive as necessary for introductory students. Finally, the interface has fixed abstractions that allow us to eliminate issues like breakpoints and different techniques for stepping through execution.

Most similar is HDPV [16], a runtime state visualization tool for C/C++ and Java programs. This work is complementary to our own in two ways. First, HDPV focuses on visualization of data structures, whereas our focus is on control flow, system state, and program behavior. Second, they deal with large programs whereas we focus on introductory courses. HDPV captures low level details like memory layout, however, they do not trace register values. For introductory students this can be quite confusing. For example, loop counters often never go beyond registers. This limitation is addressed in Frances-A by modeling registers, stack, and heap separately. Finally, Frances-A allows the user to step backward in the code. HDPV does not. Also similar is IBM's Jinsight tool [20] which focuses on more advanced topics.

GOALS FOR FRANCES-A

Specific goals motivated the design of 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.

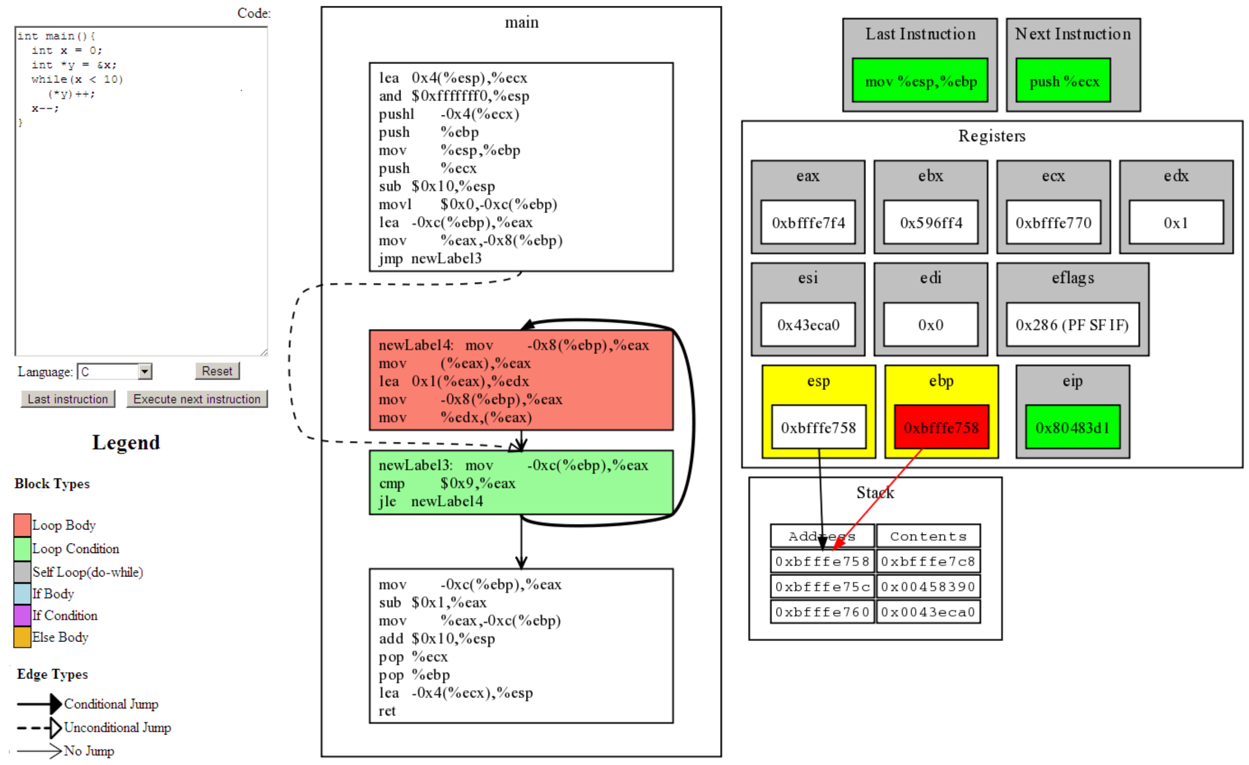
Several steps were taken to make the tool as easy to use as possible. First, the tool requires no setup, avoiding issues such as software dependencies. This helps avoid adoption hurdles regarding reliability and eases tool investigation by potential educators. Next, the simple interface design has a small 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, graphical features showing complex properties such as pointers, changes in state and accesses to memory locations are utilized. This includes a logical layout of the system state. Another highly important goal is to not require students to have a thorough knowledge of machine language to start using the tool. Since architecture courses are often the first exposure a student has to machine language, the tool needed to ease this process as much as possible.

A main goal of the Frances-A system to set it apart from others is ease of use, however, effectiveness is critical. It has been shown that the way students interact with visualization tools is more important than the visualization itself [8]. Thus, another goal is to have a hands-on tool that improves the learning process with the following properties:

1. *Support visualization on a real architecture and instruction set.* This makes the knowledge gained by using the 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 helps students 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.
3. *Allow students to step both forward and backward during program visualization.* This is a feature which is rare amongst such visualization tools. This feature is crucial to allow students to revisit complex instructions and sequences of instructions without re-running the entire simulation.
4. *A graphical and logically organized layout.* We desired a graphical layout that was logically organized, color coded to show accesses and modifications to the machine state, and illustrated pointers.

FRANCES-A

Now the major features of Frances-A are described. To start using the tool readers may point their WWW browser to the URL `http://cs.iastate.edu/~sapha/frances`.



```
int main(){
  int x = 0;
  int *y = &x;
  while(x < 10)
    (*y)++;
  x--;
}
```

```
lea  0x4(%esp),%ecx
and  $0xfffffff0,%esp
pushl     -0x4(%ecx)
push      %ebp
mov       %esp,%ebp
push      %ecx
sub  $0x10,%esp
movl      $0x0,-0xc(%ebp)
lea  -0xc(%ebp),%eax
mov       %eax,-0x8(%ebp)
jmp  newLabel3
```

```
newLabel4:  mov       -0x8(%ebp),%eax
mov       (%eax),%eax
lea  0x1(%eax),%edx
mov       -0x8(%ebp),%eax
mov       %edx,(%eax)
```

```
newLabel3:  mov       -0xc(%ebp),%eax
cmp       $0x9,%eax
jle  newLabel4
```

```
mov       -0xc(%ebp),%eax
sub  $0x1,%eax
mov       %eax,-0xc(%ebp)
add  $0x10,%esp
pop  %ecx
pop  %ebp
lea  -0x4(%ecx),%esp
ret
```

Last Instruction: mov %esp,%ebp
Next Instruction: push %ecx

eax	ebx	ecx	edx
0xbfffe7f4	0x596ff4	0xbfffe770	0x1

esi	edi	eflags
0x43eca0	0x0	0x286 (PF SF IF)

esp	ebp	eip
0xbfffe758	0xbfffe758	0x80483d1

Address	Contents
0xbfffe758	0xbfffe7c8
0xbfffe75c	0x00458390
0xbfffe760	0x0043eca0

Figure 1: Simple while loop running through Frances-A.

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, 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 point, the code entry box becomes read-only and the "Compile" button is replaced with buttons for stepping backwards and forwards in the assembly code. For example, in Figure 1 the "Compile" button has 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 pressed.

Low-level code In the middle is the machine code. This representation comes from previous work on the Frances tool [15]. This component is discussed in more detail in the next section.

Machine state Finally, the far right is the machine state. Currently, this portion of the tool supports the x86 architecture. This portion of the interface is described in detail later.

High-level to machine language relation

The middle section of the interface shows a graphical representation of the machine code (from previous work on the Frances tool targeting teaching code generation [15]). This representation allows users to easily see how high-level code maps to machine code using graphical features such as color coding and different edge drawing techniques. In Figure 1 the middle portion shows the simple while loop. The legend in the bottom left describes edge types and color codes, allowing students to easily identify the code constructs.

The purpose of this portion of the interface is to ease the burden when learning machine language. Students enter code in familiar high-level language, then see how their code is represented in machine code and how that code modifies the system state. Thus students write in familiar high level language and Frances-A provides the connection between the high level, assembly and machine states. 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. Interested readers should refer to the original paper on the tool for a thorough discussion [15].

Graphical layout of machine state

The major extension to the previous work [15] that sets Frances-A apart is the graphical representation of the machine state. In this section each aspect of the graphical representation of the machine state is discussed.

The first part 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 the last instruction placed the value in `%esp` into `%ebp`. By inspecting the current state, the user can see that these two registers contain the same value. Next, they can try to determine the effects of the next instruction before it executes.

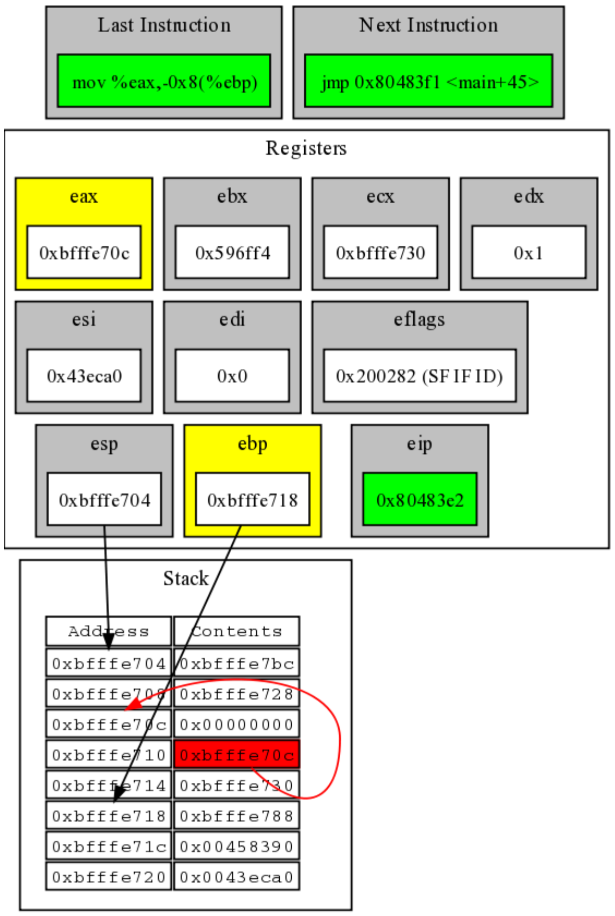
Note that the "Next Instruction" box contains the actual next instruction, not just the next sequential instruction. That is, if the "Last Instruction" was a conditional jump and the branch is set to be taken, the "Next Instruction" is the target of the branch not the fall-through case.

Next, the system's registers are separated from the rest of the state. Within this group there are logical separations. In Figure 2, notice that the first row of registers are the general purpose registers `%eax`–`%edx` 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. Also in this row is the `eflags` register which contains the results of compare instructions as well as other secondary results of operations. In this figure, the `PF`, `SF`, and `IF` flags are set (all others are unset). The user is also shown the actual hexadecimal value of this register. In the final row, there are three pointer registers. The first two are stack pointers, `%esp` and `%ebp`. 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.

Next, consider the representation of the stack. Figure 2, shows a stack of 8 elements. Each element has its own row with columns specifying the address and contents of that location. The stack is important since it contains most local variables (though some never leave the registers) as well as other temporary values. For example, in Figure 2, the last instruction moved the value in `%eax` to the fourth stack location. This corresponds to the assignment of the address of variable `x` to variable `y` from the code in Figure 1. As a result, there is now an edge from the stack location containing this value to the stack location corresponding to variable `x` (third stack location). This illustrates the behavior of pointers in the machine. 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 Figure 2 there are two edges from registers to the stack, for `%ebp` and `%esp`. When stepping through the simulation, 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 `%esp` points to the top of the stack. This helps illustrate the purposes of the `%esp` and `%ebp` registers.

Color coding in Frances-A helps illustrate the purposes of the instructions and their impact on the machine state. Green boxes denote portions of the state that always change and are not referenced directly (previous and next instructions as well as instruction pointer). Yellow boxes around the register contents signify that the last instruction accessed these registers. For example, in Figure 2, the `%eax` register has just been read (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 indicates that the contents of the register or stack has been changed by the last instruction. In Figure 1 the `%ebp` register has been modified to contain the value from `%esp` and is thus colored red. In Figure 2, the value in `%eax` was assigned to the stack location corresponding to variable `y`. 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. Similarly, consider the edge color coding. 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 were changed in the last step.



Address	Contents
0xbfffe704	0xbfffe7bc
0xbfffe708	0xbfffe728
0xbfffe70c	0x00000000
0xbfffe710	0xbfffe70c
0xbfffe714	0xbfffe730
0xbfffe718	0xbfffe788
0xbfffe71c	0x00458390
0xbfffe720	0x0043eca0

Figure 2: Frances-A after running the tenth instruction (`int *y = &x;`) from Figure 1.

Reverse stepping

Another important feature of Frances-A which is rare among similar tools is the ability to step backwards through execution. This is 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. Often students are surprised by the results of an instruction or group of instructions. The ability to step back in the instruction sequence provides immediate comparative results.

Backend

First, the original Frances tool [15]is used to display the machine code portion of the interface. Next, we use the GDB debugger [6] to perform much of the simulation. Also several newly developed programs to detect changes in state, detect pointer targets, and arrange the state appropriately are utilized. Finally, GraphViz DOT [9] is used to create the visualization.

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 that it is easy to learn, easy to use, and requires no setup. Further, several steps are taken to enhance its effectiveness. This includes logical separation of components of the machine state (including register types), edge drawing to show pointer targets and stack behavior, color coding to show accesses and writes to illustrate each instruction, and the ability to step both backwards and forwards though execution.

Future work involves support for additional architectures such as MIPS. Due to the existing design of Frances-A, this may quickly be done if users desire. Finally, a thorough evaluation (currently in progress) to ensure the usability and effectiveness of the tool will be completed. Initial results are promising.

Acknowledgments 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/ComputerScience2008.pdf.

[2] B. Nikolic *et al.* A survey and evaluation of simulators suitable for teaching courses in computer architecture and organization. *IEEE Transactions on Education*, 52:449 – 458, 2009.

[3] P. Borunda, C. Brewer, and C. Erten. GSPIM: graphical visualization tool for MIPS assembly programming and simulation. *SIGCSE Bull.*, 38(1):244–248, 2006.

[4] G. Braught and D. Reed. The knob & switch computer: A computer architecture simulator for introductory computer science. *J. Educ. Resour. Comput.*, 1(4):31–45, 2001.

[5] M. J. Conway and R. Pausch. Alice: easy to learn interactive 3D graphics. *SIGGRAPH Comput. Graph.*, 31(3):58–59, 1997.

[6] Free Software Foundation. GDB: The GNU Project Debugger. http://www.gnu.org/software/gdb/.

[7] N. Graham. *Introduction to computer science (3rd ed.).* West Publishing Co., 1985.

[8] C. 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.

[9] J. Ellson *et al*. Graphviz - open source graph drawing tools. *Graph Drawing*, 2001.

[10] M. McNally *et al.* Supporting the rapid development of pedagogically effective algorithm visualizations. *Journal of Computing Sciences in Colleges*, 23(1):80–90, 10/2007.

[11] L. Null and J. Lobur. MarieSim: The MARIE computer simulator. *J. Educ. Resour. Comput.*, 3(2):1, 2003.

[12] B. A. Price, I. S. Small, and R. M. Baecker. A taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4:211–266, 1992.

[13] P.S. Coe *et al.* An integrated learning support environment for computer architecture. In *WCAE-3 '97*.

[14] D. Sanders and B. Dorn. Jeroo: a tool for introducing object-oriented programming. In *SIGCSE*, 2003.

[15] T. Sondag, K. L. Pokorny, and H. Rajan. Frances: A tool for understanding code generation. In *SIGCSE*, 2010.

[16] J. Sundararaman and G. Back. HDPV: interactive, faithful, in-vivo runtime state visualization for C/C++ and Java. In *Symposium on Software visualization*, 2008.

[17] T.L. Naps *et al.* Exploring the role of visualization and engagement in computer science education. In *ITiCSE-WGR*, pages 131–152, 2002.

[18] U. Wolz *et al.* 'scratch' your way to introductory cs. In *SIGCSE*, 2008.

[19] J. Urquiza-Fuentes and J. Á. Velázquez-Iturbide. A survey of successful evaluations of program visu-

alization and algorithm animation systems. *Trans. Comput. Educ.*, 9(2):1–21, 06/2009.

[20] W.D. Pauw *et al.* Visualizing the execution of java programs. In *Revised Lectures on Software Visualization, International Seminar*, pages 151–162, 2002.