A

Frances: A Tool For Understanding Computer Architecture and Assembly Language

TYLER SONDAG, Intel Labs†
KIAN L. POKORNY, McKendree University
HRIDESH RAJAN, Iowa State University

Students in all areas of computing require knowledge of the computing device including software implementation at the machine level. Several courses in computer science curricula address these low-level details such as computer architecture and assembly languages. For such courses, there are advantages to studying real architectures instead of simplified examples. However, real architectures and instruction sets introduce complexity that makes them difficult to grasp in a single semester course. Visualization techniques can help ease this burden, unfortunately existing tools are often difficult to use and consequently difficult to adopt in a course where time is already limited. To solve this problem, we present Frances. Frances graphically illustrates key differences between familiar high-level languages and unfamiliar low-level languages and also illustrates how familiar high-level programs behave on real architectures. Key to this tool is that we use a simple web interface that requires no setup, easing course adoption hurdles. We also include 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. We have used Frances in courses and performed experimental evaluation. Our experiences with Frances in the classroom demonstrate its usability. Most notably, in our experimental setting, students with no computer architecture course experience were able to complete lessons using Frances with no guidance.

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, Compiler, Architecture

Additional Key Words and Phrases: Frances, Visualization, Architecture, Code Generation, Compilers

ACM Reference Format:
Sondag, T., Pokorny, K.L., Rajan, H. 2011. Frances: A Tool For Understanding Computer Architecture and Assembly Language ACM V, N, Article A (January YYYY), 28 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Fundamental to computing are the concepts of software and hardware. Most computer science courses concentrate on a high level of abstraction. Introductory programming courses focus on high-level languages and their abstractions. Likewise, algorithm courses are often implementation independent to focus on core mathematical principles. These abstractions are crucial to reducing

---

The work described in this article is a revised and extended version of our papers presented at SIGCSE 2010 and CCSC 2011. Sondag and Rajan were supported in part by the US National Science Foundation under grants CNS-06-27354, CNS-07-09217, CCF-08-46059, and CCF-11-17936.
† This article describes work that was done while at Iowa State University.
Author's address: T. Sondag, Intel Corporation, 2200 Mission College Blvd, Santa Clara, CA 95054.
K.L. Pokorny, Division of Computing, McKendree University, 701 College Road, Lebanon, IL 62254.
H. Rajan, Dept. of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, IA 50014.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© YYYY ACM 0000-0000/YYYY/01-ARTA $10.00
DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

the complexity of the problems at hand and help students understand the material and core concepts in a semester's time. However, students in any field of computing must have an understanding, at various levels of abstraction, of the computing devices and the software programs that run them.

1.1. Background

There are typically two courses that are critical in helping students understand the connections between software and hardware: computer architecture (organization) and compiler (language) design.

1.1.1. Computer Architecture. The 2008 curriculum revision for computer science states, "A professional in any field of computing should not regard the computer as just a black box that executes programs by magic... Students need to understand computer architecture in order to make best use of the software tools and computer languages they use to create programs [CC 2008, pp. 49]." The study of a computer architecture, its behavior, and instruction set are typical components to a computer organization and architecture course.

1.1.2. Language Implementation. Compiler and language implementation courses appear in all revisions of the ACM Computing Curricula. The latest revision of CC '01 states the importance of these topics: "...good compiler writers are often seen as desirable; they tend to be good software engineers [CC 2008, pp.11]." These compiler design and programming language courses typically involve language translation from high to low-level languages (typically an assembly language).

Aside from these courses, while most students will never write assembly code directly, understanding the code generated by compilers is important for both programming and debugging tasks.

1.2. Problems

In the past, courses on architecture, compilers, and programming language design were often supported by a prerequisite chain containing courses on assembly and low-level programming languages. However, in recent years these prerequisite courses have slowly been supplanted by other topics in many undergraduate computing curricula [Loui 1988]. As a result of this typical curricula design, students have little exposure to these low-level details when they reach courses like computer architecture or compiler design. Thus, not only must students learn these new topics, which differ from anything they have learned before, they must also continue to learn other complex topics introduced in these courses (e.g. architecture, language design, etc.). Therefore, there is a real need to enhance methods and techniques for teaching students assembly language and low-level programming topics.

For learning and teaching these difficult topics, it is desirable to use interactive tools that have been shown to be highly effective for education [Naps et al. 2002; Hundhausen 2002]. Many tools exist for visualizing programs [Aiken 1996; Corliss and Lewis 2008; Resler and Deaver 1998] and simulating programs on different architectures [Coe et al. 1995; Nikolic et al. 2009; Null and Lobur 2003; Graham 1985; Braught and Reed 2001; Borunda et al. 2006]. However, they are typically time consuming to learn and difficult to use. As a result, adopting them in a course is challenging. Especially in the presence of all the other new material that must be learned.

1.3. Contributions

To solve these problems, we present the *Frances* tool[1]. A major idea behind our approach is to take advantage of the students' existing knowledge of high-level language programming. Thus, the Frances tool allows students to enter familiar high-level code that is then shown alongside a graphical representation of the assembly code and machine state. We make use of several novel features to help students make connections between high-level programming languages, assembly language, and the low-level architectural concepts of the computing device. In summary, Frances

[1]We named the tool Frances in honor of Frances E. Allen . She received the Turning award for pioneering contributions to the theory and practice of optimizing compiler techniques that laid the foundation for modern optimizing compilers and automatic parallel execution.

— presents a visualization of the low-level code that maintains actual target *code ordering*,
— differentiates between *types of run-time paths* in the low-level code,
— *color codes* instruction blocks by their high-level control constructs,
— shows how each individual machine *instruction impacts the machine state*,
— displays the components of the system state in a *logical organization*, illustrating several important concepts,
— allows for both *forward and backward stepping* through program steps, which allows students to revisit complicated steps and processes,
— *color codes* individual parts of the machine state making the impact of each instruction clear,
— clearly illustrates difficult concepts surrounding addresses (e.g. pointers and stack) using *color coded arrows*.

Thus, we have designed Frances to help students understand low-level languages, language translation (code generation), and computer architecture by showing how familiar high-level code maps to low-level code and how that low-level code behaves on a target architecture.

1.3.1. Ease of Adoption. We have developed Frances in a way such that it is platform independent, requires no setup, and is trivial to begin using. By providing our simple web-based interface that is easy to learn, we avoid three of the four biggest factors hindering adoption of such visualization tools in educational settings[2] 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) [Naps et al. 2002]. We believe that all of this together makes Frances easy to adopt, use, and understand.

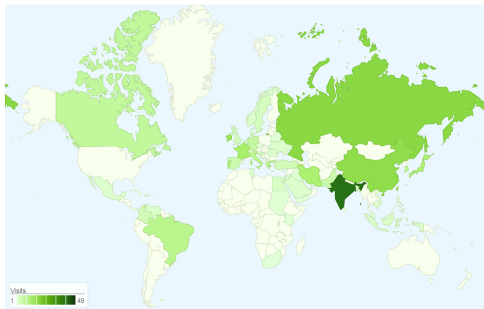

Fig. 1. Worldwide usage of Frances between April 2009 (initial release) and January 2012. USA traffic is excluded.

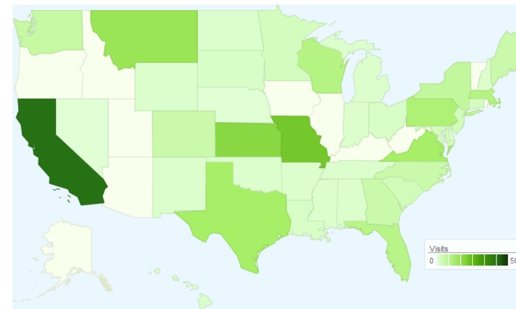

Fig. 2. Frances usage within the US since between April 2009 (initial release) and January 2012. Traffic from the authors and their students is excluded.

To further facilitate adoption, Frances is freely disseminated for educational purposes (`http://frances.cs.iastate.edu`) and at this time we know of several educators around the world that have tried out our tool since its public release showing its potential. Figures 1 and 2 show use of the Frances tool between its public release in April 2009 until January 2012 (excluding use by the authors and their students). Figure 1 shows a world map where countries highlighted with green represent those trying out our Frances tool (this figure excludes traffic from USA). Figure 2 shows a map of the USA where states highlighted with green represent those trying out our Frances tool (this figure excludes traffic from the authors and their students). These figures help demonstrate the wide dissemination, usefulness, and ease of adoption of Frances.

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

1.3.2. Usefulness. We have used Frances in three undergraduate courses and have performed evaluations of the tool, to determine its ease of use and usefulness, independent of any particular course. Overall, we observed encouraging results. Students found the tool useful for learning topics regarding code generation and often used it as a reference when implementing their own compilers. Based on this experience, Frances has been made a more integral part of these courses. Experimentatal evaluation of Frances in terms of ease of use and usefulness has shown promise. Even allowing students who have never had a computer architecture course to complete the assignments they were given with no guidance.

The rest of this paper is organized as follows. Section 2 presents related ideas. Next, Section 3 describes our goals when developing the components of Frances. Section 4 describes design and implementation of the Frances tool. Section 5 describes our experimental results and course experiences. Finally, Section 6 discusses future work and concludes.

2. RELATED WORK

Related work can be classified into three major categories: teaching assembly language and language translation, simulating architectural details and teaching computer architecture, and visualizing programs. We now compare Frances to related work in each of these categories.

2.1. Teaching Assembly Language and Low-Level Language Translation

Since developing a compiler is difficult, especially within a single semester course, a large body of work has been done to improve this process. Related work in this area can be broadly classified into two categories. The first category is related work dealing with tools designed to help students build and learn about compilers / language translation systems [Aiken 1996; Corliss and Lewis 2008; Resler and Deaver 1998; Gondow et al. 2010]. Work in the second category deals with teaching assembly language [Zilles 2005; Bredlau and Deremer 2001].

Gondow *et al.* developed MieruCompiler [Gondow et al. 2010] that is used for visualizing various pieces related to the source code such as assembly code, abstract syntax tree, symbol table, stack layout, etc. Our approach differs in that we provide a more visual representation of the assembly code.

Aiken presented Cool [Aiken 1996], a language and compiler designed for course projects to reduce the overhead for the instructor and keep assignments modular. Similarly, Corliss and Lewis developed Bantam [Corliss and Lewis 2008], a Java compiler project for courses. Modularity is also achieved in Bantam since components of the compiler can use the provided modules, or be swapped out with custom versions. Rather than developing a new infrastructure, our technique is complementary to these existing techniques in order to help understand specific portions of compiler implementation.

Resler and Deaver propose a visualization tool, VCOCO [Resler and Deaver 1998], for understanding compilers. VCOCO provides several view panes that show source code, language grammar, compiler, parser, and scanner. Each pane is updated throughout the compilation process. We also propose a visual approach, however, we are interested specifically with code generation and present a graphical approach.

Zilles developed SPIMbot [Zilles 2005]. SPIMbot provides an environment where learning assembly is put to use for programming virtual robots. Our approach takes a more traditional approach that is likely to have less overhead.

Bredlau and Deremer suggest using the Java Virtual Machine (JVM) for teaching assembly [Bredlau and Deremer 2001]. The idea is to let the Java compiler create JVM code that is compared to the source code. We take a similar approach but with assembly language and we provide a graphical comparison to aid in this process.

2.2. Architectural Simulators

A large body of work exists for simulating architectures and teaching computer organization and architecture [Schwetman 1986; Pai et al. 1997; Coe et al. 1995; Kise et al. 2004; Zeng et al. 2009;

Binkert et al. 2003; Nikolic et al. 2009; Null and Lobur 2003; Graham 1985; Braught and Reed 2001; Borunda et al. 2006]. 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 [Null and Lobur 2003] for use in teaching computer architecture and assembly language. This approach has several advantages including a simple assembly language and an accompanying text book. Similarly, Patt and Patel developed LC-3 [Patt and Patel 2004] which makes use of a simple instruction set that is similar to x86. LC-3 also has an acompanying text book. However, some argue that there are advantages to using real assembly languages rather than custom languages [Gondow et al. 2010]. This is a fundamental difference in our approaches. For those who prefer to use a real assembly language we recommend Frances. Further, our work differs in several ways. First, with MarieSIM and LC-3 there is a disconnect between high-level languages and the low-level instruction set since users must program simulations in the low-level language. With Frances, 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 and LC-3 have accompanying text books. 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 into existing courses, we develop our course materials around topics that complement existing courses utilizing a standard textbook. Further, Frances is released via the web rather than requiring installation, which hinders adoption because of compatibility and dependence issues. Finally, MarieSIM and LC-3 do not allow "stepping" backwards through program execution. Frances has this feature to allow students to revisit previous execution steps without re-running the entire simulation.

Graham developed "The Simple Computer" simulator [Graham 1985]. Stone later used this simulator to teach CS1 topics [Stone 2006]. Another simulator developed by Braught and Reed is targeted toward introductory students in CS0 [Braught and Reed 2001]. The main differences between these works is that (a) Frances has a graphical interface that we believe has a much lower adoption time, and (b) Frances uses real instruction set architectures (ISA) rather than custom machines and languages.

Borunda *et al.* developed GSPIM [Borunda et al. 2006], a MIPS simulator, for use in introductory computer architecture courses. This tool shows simultaneous views of the program call graph, intraprocedural control flow graph, MIPS assembly code, and registers. Our work differs in that Frances 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 will be more effective when learning assembly language.

2.3. Program Visualization

A large body of work exists for software visualization [Price et al. 1992; McNally et al. 2007; Urquiza-Fuentes and Velázquez-Iturbide 2009; Rössling and Velázquez-Iturbide 2009; Esponda-Arguero 2008; Hundhausen 2002; Naps et al. 2002; Sundararaman and Back 2008; Pauw et al. 2002; Reiss 2003; Conway and Pausch 1997; Sanders and Dorn 2003; Wolz et al. 2008; Powers et al. 2006]. Price *et al.* developed a taxonomy for software visualization [Price et al. 1992]. 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. In terms of program visualization, a major difference between our work and previous work is that our interface is much simpler and more straightforward. Thus, we address the primary factor limiting adoption of previous work [Naps et al. 2002]. We are able to do this because Frances's backend consists of several powerful program analysis techniques.

Rather than requiring installation, Frances is deployed via a web interface thus removing additional hurdles such as software and OS dependencies. Therefore, we avoid many of the factors that hurt adoption of such tools [Naps et al. 2002]. Finally, we developed Frances around a real machine and instruction set rather than custom models thus avoiding a disconnect between the tool and real architectures [Gondow et al. 2010].

An example of a program visualization technique is a debugger such as gdb [Free Software Foundation 2010], or a graphical debugger such as kdbg [Sixt 2010], 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 incompatible 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 our work is the work by Sundararaman and Back on HDPV [Sundararaman and Back 2008], 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 machine model that captures low-level details like memory layout. Since the focus of HDPV is on data structures, it does 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 that can be compiled to native code. To enable several low-level implementations, HDPV makes use of Pin [Luk et al. 2005]. Similarly, we make use of cross-compilers and GNU BinUtils [Free Software Foundation 2009] to support a variety of targets. Frances allows the user to step backward in the code. HDPV does not.

Also similar is IBM's Jinsight tool [Pauw et al. 2002]. 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 a web-based approach. A tool similar to Jinsight was also developed by Reiss [Reiss 2003]. This tool, like Jinsight, is targeted towards more experienced programmers and looks at phase behavior [Sherwood et al. 2002] and performance issues.

3. DESIGN CRITERIA FOR FRANCES

In this section, we discuss the design criteria for developing the Frances tool as well as how we accomplish these criteria. Briefly, these criteria include making various low-level aspects of computing easy to understand for students such as assembly language, computer architecture, and code generation. We do this by clearly and quickly showing how familiar high-level language constructs translate to low-level language code and how this code behaves on the hardware. Additionally, we want Frances to be easy to learn, require no setup, and not require thorough knowledge of a machine language or computer architecture to get started. Further, we want it to use real architectures and instruction sets. Next, the tool needs to be effective. To ensure this, we desire a tool that would allow students to clearly visualize the behavior of each machine instruction and how familiar high-level code maps to this machine code.

3.1. Ease of Use

We felt it was important for Frances to be as easy to use as possible. If not, the cost of learning how to use Frances could easily overshadow the benefits it provides.

We took several steps to make Frances easy to use:

— require no building, setup, or installation,
— provide a simple graphical interface with little learning curve,
— support a variety of high-level languages, and
— support a variety of low-level languages.

This is all necessary to ensure that Frances is feasible for adoption in a single semester without distracting students or drastically changing existing courses.

Because we do not require users to build or set up Frances on their systems, it is faster to begin using; issues such as software, hardware, or OS dependencies are avoided; and reliability is improved. All that is required is a web browser. Users may even access the system on simple devices such as cell phones or tablets, which are rapidly gaining popularity.

Many related tools operate from the command line and require the users to learn complicated syntax. While powerful and flexible, learning how to use such a tool if it is only planned on being used for a short period of time is undesirable. We believe our interface is simple enough for users to immediately begin using it and understanding the output. To facilitate this simple interface, we make use of several more complicated analysis techniques on the backend. Further, we rely on graphical features to show complex properties such as control flow, pointers, changes in state, accesses to memory locations, etc. This includes a simple logical layout of the system state.

An assumption of our system is that students are familiar with a high-level language. Thus, we have designed our system to be as easy as possible to integrate a variety of high-level languages.

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. Usefulness

One of the main goals of our system to set it apart from others is ease of use, however, usefulness is still critical. Without it, ease of use is pointless. It has been shown that the way students interact with visualization tools is more important than the visualization itself [Hundhausen 2002]. Thus, we want a hands-on tool to aid in the learning process. To make this hands-on tool, Frances, useful, we had several goals.

(1) *Allow students to step both forward and backward during program visualization*. This is a feature that 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, which can cause the student to lose context.
(2) *Illustrate key differences between high and low-level languages.* We have observed that for many students, learning low-level languages is difficult and code generation can be the most difficult challenge when writing a compiler for the first time. This is largely because of the differences in already familiar high-level languages and unfamiliar low-level languages. This includes differences in syntax as well as the ordering of statements related to the various programming constructs. For example, Figure 5 shows how the order of the loop condition and loop body are opposite in the two representations.
(3) *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.

(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.
(5) *Support visualization on a real architecture and instruction set.* This would make the knowledge gained by using Frances applicable to standard learning materials and in the real world (e.g. real languages and real architectures).

4. FRANCES

We now describe the use and major features of Frances. This includes the simple interface to the tool and the two major components of the tool. To start using the tool readers may point their WWW browser to the URL `http://frances.cs.iastate.edu`.

We now describe the main components of Frances in detail. This includes the code entry and control of the system, representation of the low-level language, and representation of the machine state.

4.1. Code Entry and Control

Key to Frances is an easy to use web-interface that requires no setup. An example of this interface is shown in Figure 3.

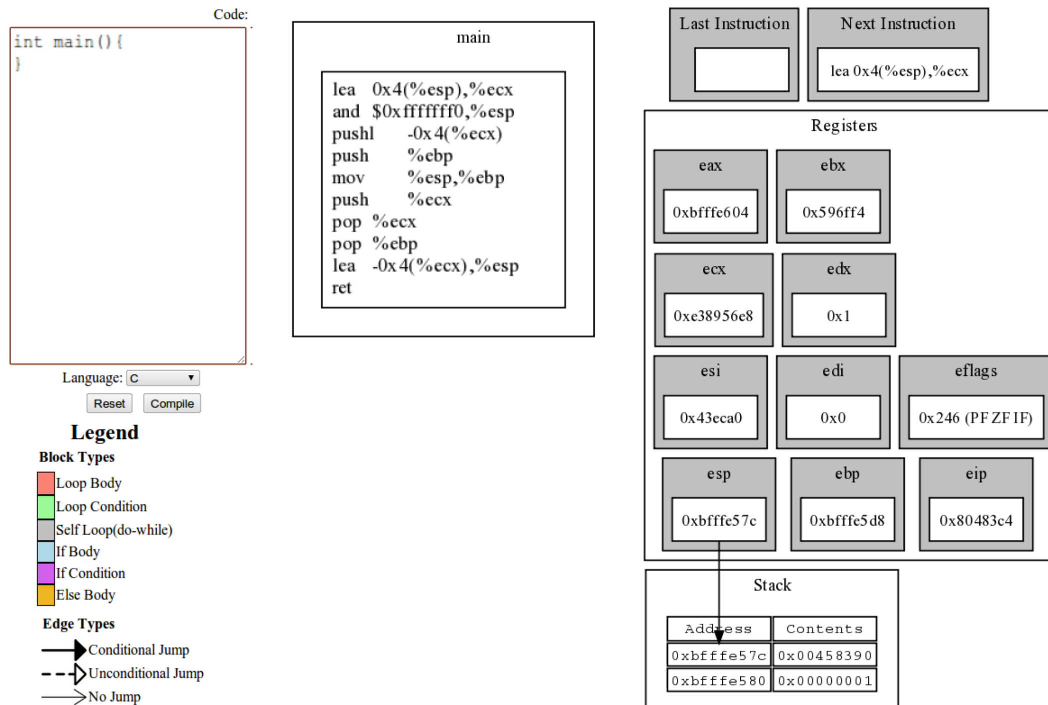

Fig. 3. Frances starting state. The left side shows the box for code entry and the compile button that, when clicked, will generate the graphical representation of the assembly code. The middle shows the graphical representation of an empty program and the right side shows the machine state prior to executing the code.

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 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.

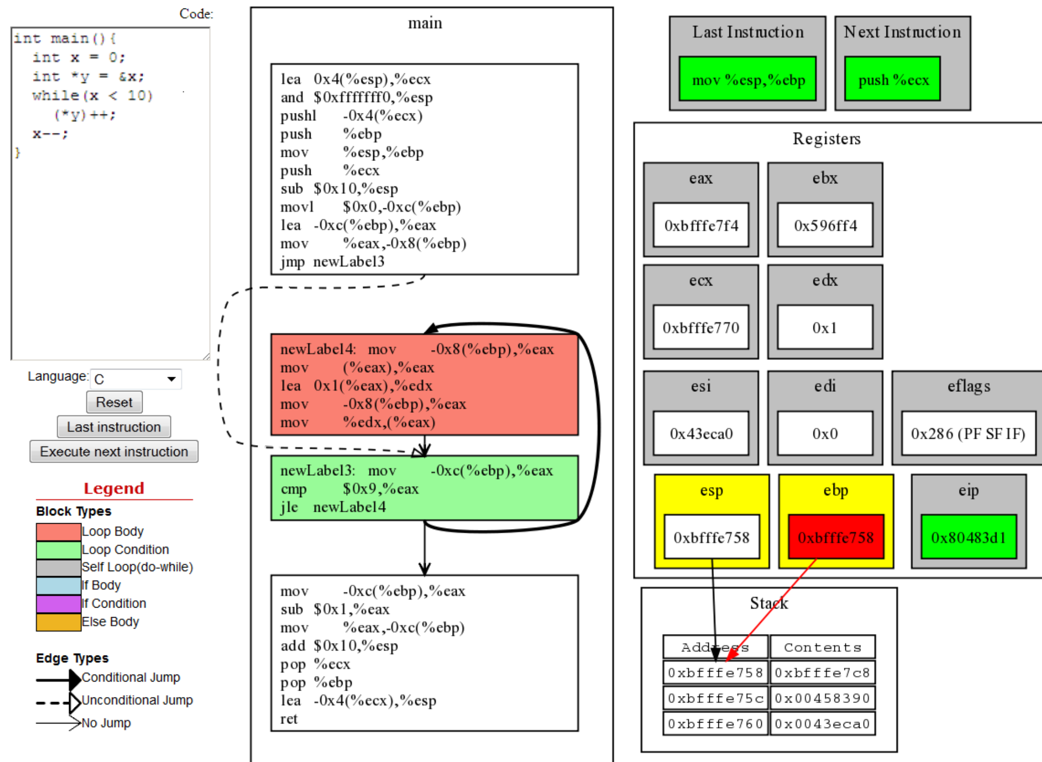

Fig. 4. Shows a simple while loop running through Frances. The high-level code has been entered on the left side and been compiled. The middle shows the graphical representation of the assembly code for this loop. The right side shows the state after the fifth machine instruction.

In Figure 4, code has been entered, 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 pushed.

Since the "Compile" button has been pressed, we can see the buttons for "Last instruction" and "Execute next instruction". As the labels suggest, these buttons control the simulation of the program and update the machine state accordingly. The behavior of buttons and how they impact the machine state is detailed in Section 4.3.

4.2. Relation Between High-level to Machine Language

In the middle of the interface is the graphical representation of the machine code. This part of the interface allows students 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. Key to our approach is illustrating the differences between already familiar high-level languages and unfamiliar low-level languages. This includes differences in syntax as well as the ordering of statements related to the various programming constructs.

The purpose of this portion of the interface is to ease the burden of learning an assembly language and/or language translation. Students may enter visualization code in a familiar high-level language, then see how their code is represented in assembly instructions and how these instructions modify

the system state. This means that students do not have to write simulation code in an assembly language which speeds up the overall educational process and helps ensure that users understand the meaning of the assembly code. Finally, it also helps the user visualize how different high-level program structures behave at the machine level.

We observe that there are differences between code ordering in high-level and low-level languages. For example, in Figure 5 the order of the loop condition and loop body are opposite in the

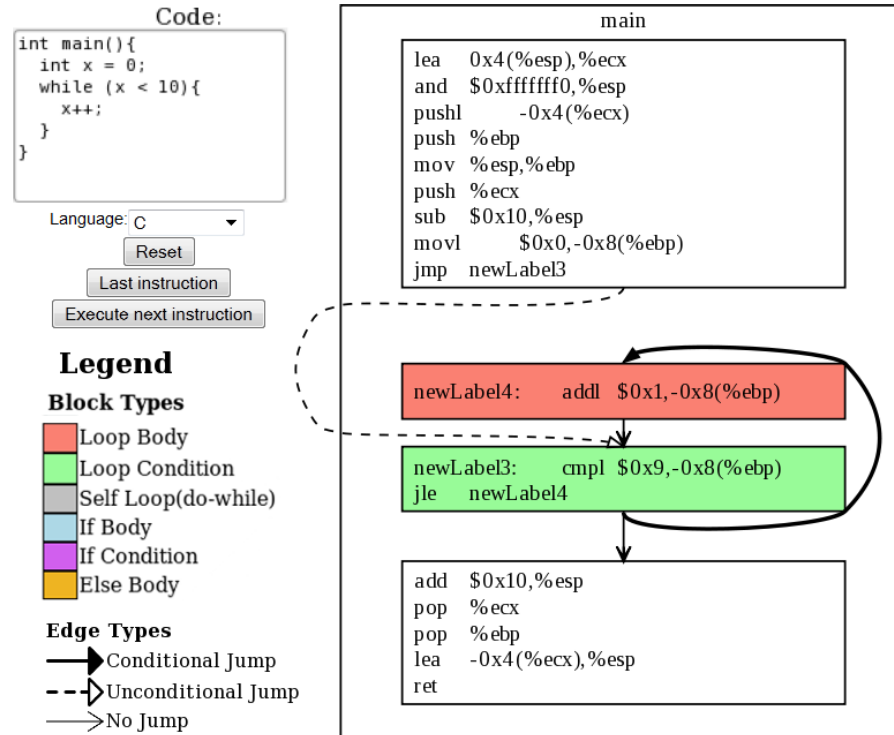

Fig. 5. A simple C while loop

two representations. With this part of the tool, we aim to ease this challenge by clearly showing how familiar high-level language code and constructs map to target, low-level assembly code. The idea is that students already understand at least one high level language. By understanding how the high-level and low-level languages relate to each other, students can quickly understand how to translate from one language to the other. In a language design or compiler design course, students can use this portion of the tool as a guide for dealing with various high-level constructs including memory management and code order.

A simple way to compare the two languages is to have students compare source code with equivalent assembly code (for example, gcc can generate assembly from source). The benefit of this is that it allows students to see what types of instructions their code is mapped to as well as the ordering of instructions. This process is helpful, however, we felt that more could be done to improve it. To improve this process, we show the source to target language mapping and improve upon this mapping in two ways. First, we represent target code as a graph that shows execution paths that may be taken at run-time. This helps students understand how different types of jumps work and the control flow of the machine code is implemented. Second, we color code this graph to relate how control structures represented in the source language are mapped to the target language. Most students are familiar with high level languages constructs. Thus, being able to quickly identify how

language constructs in familiar high-level languages map to assembly code significantly eases this comparison process. The details of these features and how they work are described in the rest of this section.

4.2.1. Backend. Key to implementing this part of the tool is a binary analysis and instrumentation framework built for the SAPHA project [Sondag and Rajan 2011; 2010]. A diagram of this framework is shown in Figure 6.

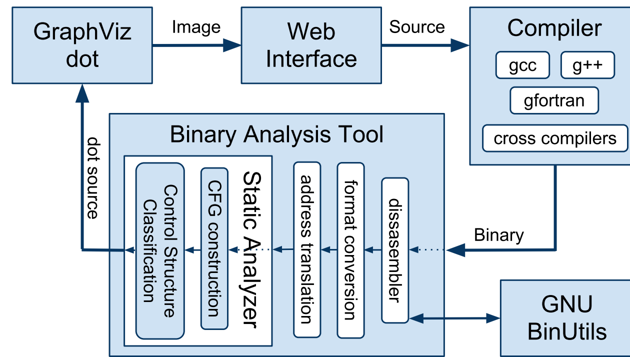

Fig. 6. Low-Level Component Architecture

Currently, this part of the tool allows users to input their own high-level language code in a variety of languages. This code is then translated by a compiler (e.g. gcc) to an object file as shown in Figure 6. Thus we are not tied to any language or compiler. Other high-level languages are also immediately supported by simply installing a compiler on the server. For example, FORTRAN support was added by simply sending the user's source code through a FORTRAN compiler instead of a C/C++ compiler.

Next, we make use of the GNU BinUtils [Free Software Foundation 2009] libraries to help convert these object files back into assembly code. The BinUtils library supports a variety of target architectures and thus makes our tool very general in terms of target architecture and high-level languages. For example, we designed the system for the AT&T x86 syntax but have added support for Intel x86 syntax by simply changing minor settings and have added MIPS support by simply building a cross-compiler on our server.

Our analysis tool then converts the assembly code into a format more suitable for analysis (C++ objects). Since the binary contains virtual addresses for calls and jump targets, we convert these into labels, which are easier to read. After address translation, we can construct the control flow graph (CFG) of the program.

The next step is done using our analysis and instrumentation framework, which consists of a variety of static analysis techniques. For this step, we use our implementations of standard control flow analysis algorithms [Allen 1970; Muchnick 1997]. Through this process, the CFG is converted to a graphical format where blocks of code are nodes in a graph. These nodes are arranged in the same order as the assembly code to demonstrate how instruction ordering of low-level code works and how it differs from high-level code. To further illustrate this topic, graph nodes (blocks of instructions) are colored to illustrate the purpose of the code (loop condition vs loop body, etc). This approach also handles nesting of control structures. Additionally, edges are drawn to illustrate potential program paths. These edges are drawn differently to illustrate the type of path (conditional jump, unconditional jump, or no jump / fall-through). The results of this step are output as a "dot" file that is sent to the GraphViz dot [Ellson et al. 2001] program, which generates the graphical representation for the interface.

The source code of this framework is not yet publicly available, but will be released as an open source project in the future. Until then, the tool is made available as a web service.

4.2.2. Graphical Representation and Interface. This component of the tool generates a simple graphical representation of the target code corresponding to the source code. For example, in Figure 5, this simple *while* loop is shown graphically as four blocks of code. Furthermore, the edges or *paths* between these blocks that can be taken at run-time are shown. To generate the graphical program representation, we make use of `dot`, which is part of the GraphViz [Ellson et al. 2001] graph visualization software. We now describe the major components of the representation including how blocks and edges are drawn as well as a brief discussion about the interface.

4.2.3. Blocks. Basic blocks of instructions are generated. A basic block is a sequence of instructions with a single entry point and single exit point with no jumps between [Allen 1970]. For simple control structures (non-nested structures) basic blocks capture the main components of the structures. For example, in Figure 5, the sample *while* loop can be divided into two parts each with a different purpose: the *loop body*, and the *loop condition*. Therefore, the graphical version of the target code illustrates these two components of the loop in two separate blocks. Additionally, the blocks before and after the loop are also shown separately.

A major difference between previous tools and our tool is the way we lay out blocks. Similar tools [The Informatik Centrum Dortmund (ICD) ; Titzer et al. 2005; AbsInt] represent blocks as a flow chart. Since our major goal was to help students understand code generation, we make this graphical representation as close as possible to real generated code. We do this by maintaining the instruction ordering of the actual target code. This includes the ordering of the blocks. To make this ordering clear, we represent blocks in a linear fashion in the same way that programs are represented in target code.

For example, in Figure 5, the layout of blocks is not done in a way that is immediately obvious from the source code. Consider the loop condition. In the source code, this is before the loop body whereas in the target code, it is after the loop body. This is not immediately clear to introductory students; however, for very good reasons this is how target code is generated by the compiler. Thus, understanding this ordering is necessary for understanding code generation. Therefore, our tool exposes students to such orderings. Given that this ordering is confusing, we take steps to help clarify this ordering.

4.2.4. Color Coding. Since students are already familiar with a high-level language, we aim to quickly and clearly illustrate how control structures in a high-level language are represented in the target language. We show this by coloring the graph to highlight the different parts of the various control structures. Our tool performs simple control flow analyses [Muchnick 1997] on the code to determine the different parts of the control structures. Then, the tool colors blocks based on the part of the control structure they make up (loop condition, loop body, etc). For example, in Figure 5 a *while* loop is shown in both forms. Both the loop condition and the loop body are colored differently to make it easy to distinguish between the two. As mentioned previously, the ordering of these two blocks is confusing at first since it differs from the source code ordering. This coloring, as delineated in the Legend, quickly points out this ordering by showing that for this high-level language *while* loop, the loop condition goes after the loop body.

For simple control structures, that is, non-nested control structures, we shade the background of the blocks to corresponding colors for each part of control structures. This includes structures such as loops (loop body and loop condition are colored differently), *if/else* blocks, etc. As discussed previously, Figure 5 shows this for a simple *while* loop.

For nested control structures, the coloring must be done differently. We start by shading the blocks in the innermost structures as described previously. Then, all other structures are surrounded with boxes. These boxes are then shaded to show what kind of structure the member blocks are a part. Furthermore, this helps to show how the different structures interact. For example, consider the nested loop in Figure 7. The inner loop is composed of two blocks, the loop condition and the loop

body. These two blocks are shaded in the figure. We can see that both of these blocks (the entire inner loop) are contained within another structure since they are contained in a larger shaded box. This structure is the loop body of the outer loop, which is shown clearly by the shading.

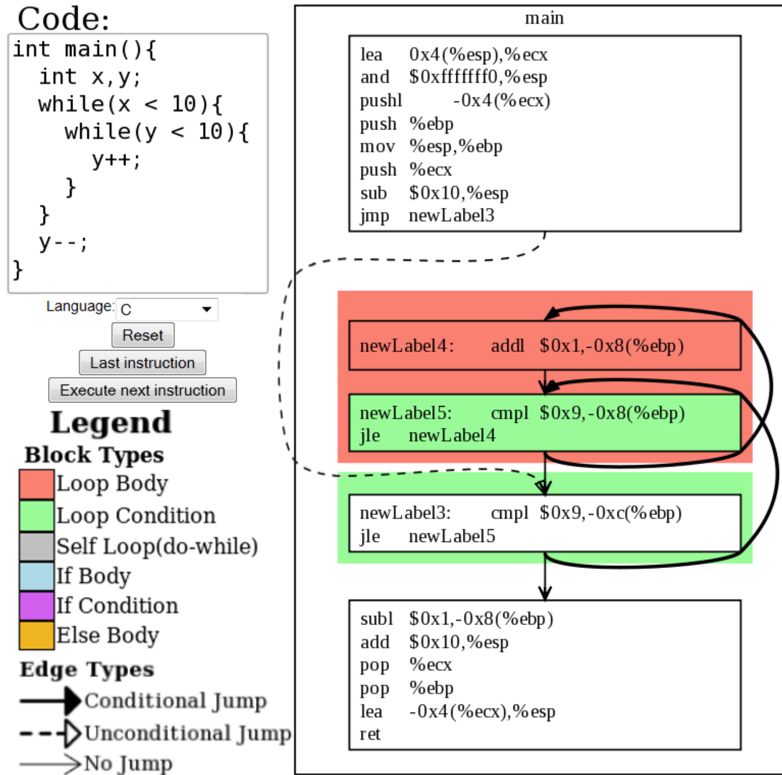

Fig. 7. A simple nested while loop

This drawing of blocks shows how target code is laid out. Then the coloring helps to quickly show how components in familiar high-level code are represented in the target code. Furthermore, by breaking this representation down, we can focus on a smaller subset of the code.

4.2.5. Paths. The edges between blocks represent the *paths* that can be taken at run-time. A jump in the target code can have up to two possible next instructions. The paths show these possible next instructions. For example, in Figure 5, we see that the loop condition (shown in green) has two outgoing paths: one edge leading to the loop body (if the condition is true – conditional jump) and one edge leading to the next block after the loop that exits the loop (if the condition is false – no jump / fall-through).

In combination with blocks, edges help the user see how different structures are represented. For example, consider the first block in Figure 5. This figure illustrates how, in the target code, execution first jumps past the loop body to the loop condition (using an unconditional jump) for this type of loop. This illustrates a key difference between *while* and *do-while* loops since *do-while* loops are not organized this way.

As mentioned previously, the instruction (and block) ordering in the target code can be confusing to students because it is frequently different from the source code ordering. Our graphical representation of blocks helps by highlighting the components of the different control structures. Illustrating the ordering of control structures is helpful; however, we still need to show execution flows between

these structures. Figure 7 shows an example of a nested loop where paths help to illustrate the initially confusing code layout. In this figure, we see that the edge corresponding to entering the inner loop actually goes to the second block in the inner loop. This is slightly confusing at first since the path does not go to the beginning of the inner loop code. This example shows that it is important to understand how execution enters and exits loops. Furthermore, understanding how execution flows through others structures such as `if /else` blocks is also important. Thus, we take steps to help contrast the differences between edges.

4.2.6. Edge Types. There are multiple types of edges. We illustrate the different types by using different styles of lines and arrowheads for drawing the edges. For example, in Figure 5, we see all three different types of edges. We now give a brief description of each edge type.

— First, we have *"unconditional jumps"*. In the figure, this jump is illustrated with a dashed line and an empty triangular arrowhead. In Figure 5, the first block ends with the instruction `jmp newLabel3`. With this type of jump, the path is taken whenever the instruction is executed.
— Next, we have *"no jump"* (or *"fall through"* / *"branch not taken"*) edges. This edge type is illustrated with a thin edge and a "wedge shaped" arrowhead. This edge type goes to the next sequential instruction when either the current instruction is not a jump or a condition is false. For example, the edge going from the loop body to the loop condition in the figure. In this case, since the block does not end with a jump, the next instruction is just the next sequential block. Another example of this type of edge is the edge from the loop condition to the last block in Figure 5. This edge is taken when the condition on the jump, in this case `$0x9 >= -0x8(%ebp)`, is false. This may seem trivial since the *"no jump"* edge is always the edge to the next sequential block, however, for students, this concept may not be immediately obvious.
— Finally, we have *"conditional jump"* (or *"branch taken"*) edges. These edges are drawn with a thick solid line and a solid triangular arrowhead. These edges are those that are taken when a jump condition is true. For example, in Figure 5, we have an edge from the loop condition block to the loop body block. This edge is taken whenever the loop condition on the loop is true. Another example is shown in Figure 8. In this example we can see a branch taken edge from the *if condition* block to the *else body* block. This edge is taken whenever the condition is true, however, in the source version, we have that this edge is taken whenever the if condition is false. This is another interesting difference between source and target code, which is nicely illustrated by this part of the tool.

This edge drawing helps illustrate the finer details of the target code. This includes how individual instructions such as jumps are created and how complex control structure components such as nested loops, interact . Together with our block drawing and coloring, this component of the tool generates informative and easy to understand figures, which illustrate how code generation is performed.

4.2.7. Benefits of Relating High-level to Machine Language in Frances. The block layout and coloring in combination with the edge drawing greatly helps to teach the instruction layout of low-level language code. Furthermore, when viewed alongside familiar source code, this representation makes the process of understanding translating between the two languages significantly easier. With its simple and easy to use interface, Frances is easy to use in a course, will help students understand these difficult concepts, and save valuable course time for other topics.

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 9, a `mov` instruction was just executed and `push %ecx` is next.

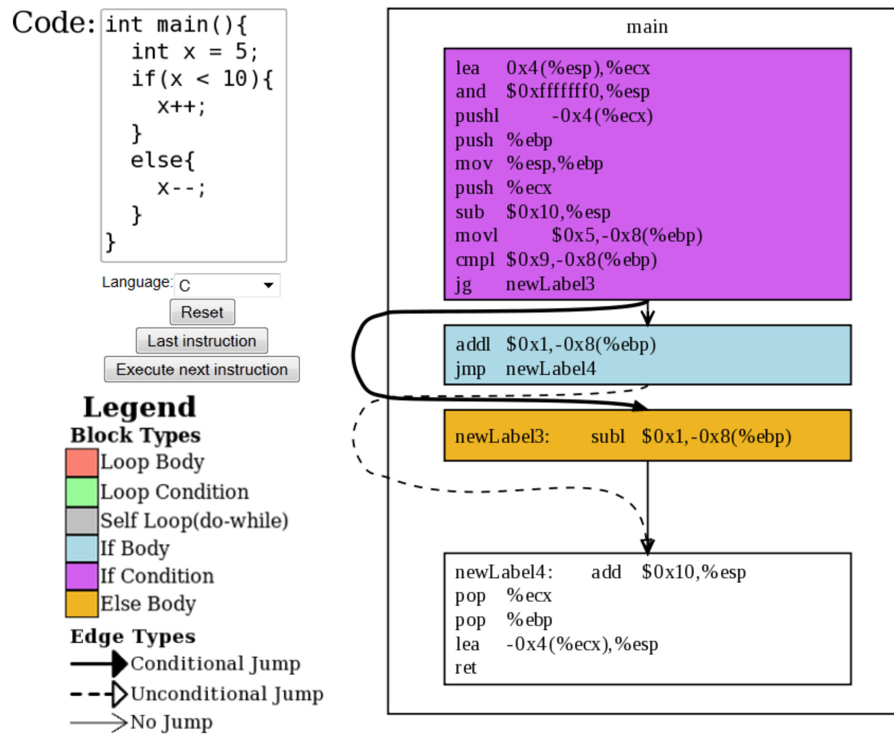

Fig. 8. if-else block

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, in Figure 9 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 that 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 10, 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 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 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 10 shows a stack containing 8 elements. Each element has its own row with columns specifying the address of the stack location

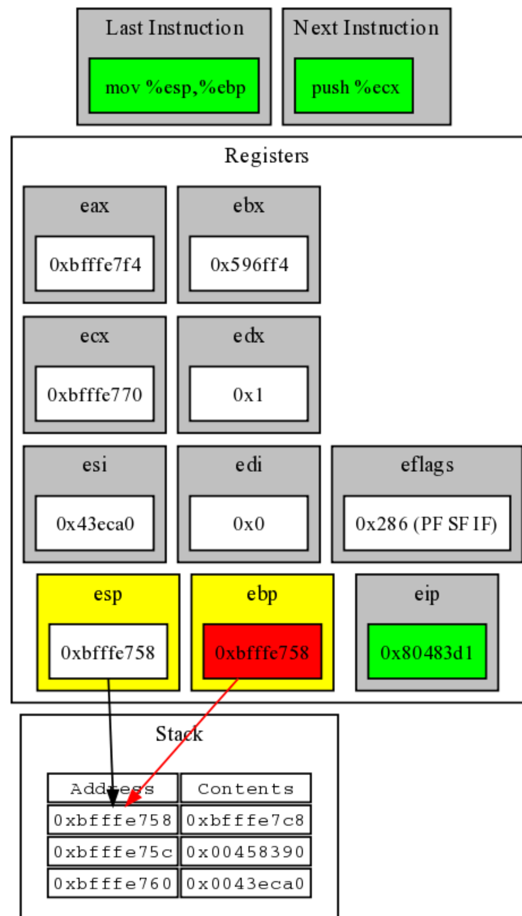

Fig. 9. Architecture state portion from Figure 4.

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 10, the last instruction moved the value 0 to the third stack location. This corresponds to the assignment of `int x = 0;` from the code in Figure 4. 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.

4.3.4. Edges. Now, let us consider the edges in the figures. First, in Figure 10 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 11. 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

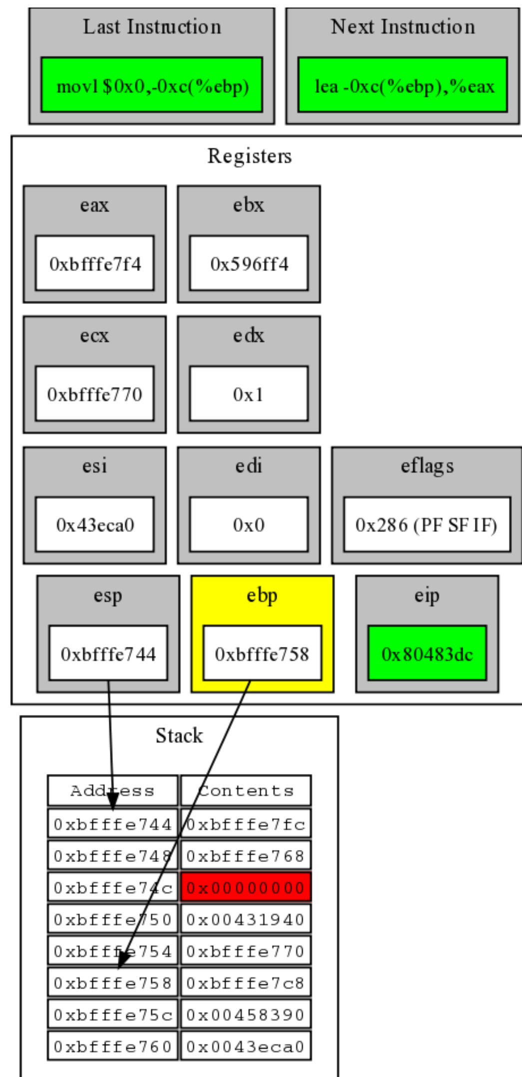

Fig. 10. Frances state running the same code example as Figure 4 after the seventh instruction (`int x = 0;`).

variable `x` (third location on the stack). This illustrates the behavior of pointers in the machine. This is an important yet difficult concept for introductory students.

4.3.5. Color coding. Until this point, we have ignored the color coding of the machine state representation in these figures. We now describe this aspect of Frances, 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.

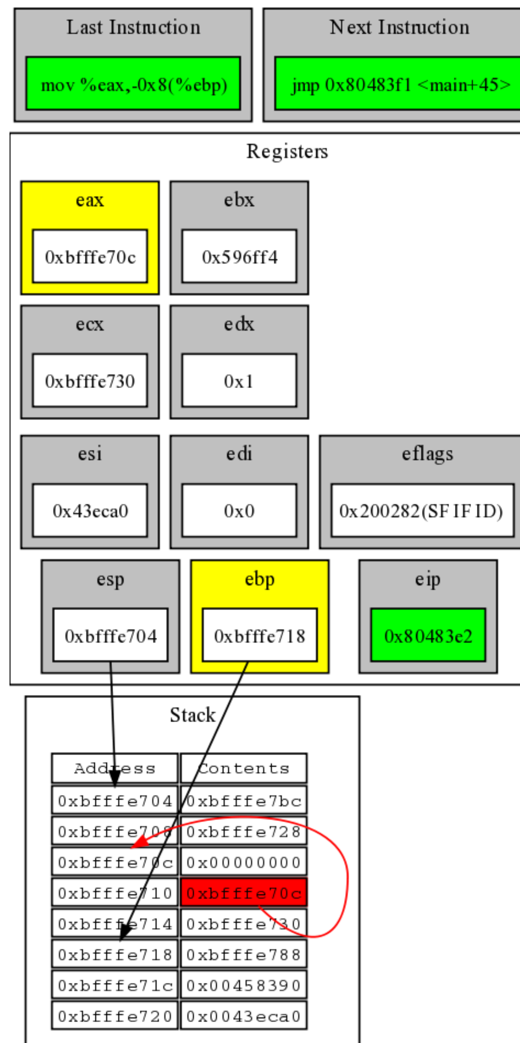

Fig. 11. Frances state running the same code example as Figure 4 after the tenth instruction (`int *y = &x;`).

Next, the color coding of the boxes for registers are considered. Yellow boxes around the register contents signify that the last instruction accessed these registers (either read or write). For example, in Figure 11, 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 were changed by the last instruction. Consider Figure 4 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 10, 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 11, 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 indicate change. In this example, since there is a modification to the pointer, the edge is red.

4.4. Reverse stepping

Another important feature of Frances, 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. With Frances a student can step back and forth to ascertain every detail of an instruction's impact on the machine state.

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 4. We can see that the `%ebp` register 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 the GDB debugger [Free Software Foundation 2010] to step through instructions individually and obtain the values in each register and stack location between each instruction. Aside from this, we use several scripts and programs to visually present the state as well as determine changes. After this, we use the GraphViz DOT program [Ellson et al. 2001] to draw the visualization of the machine code and machine state. Each visualization state's image is stored on the server until the user is done with the visualization ("reset" is clicked or a timeout occurs). Thus, to implement backwards stepping, we simply load the image associated with the previous state. Similarly, when forward stepping to a state already generated, the server can simply load the images for these states.

5. EVALUATION

In this section we discuss the evaluation of the Frances tool. First, Frances is compared with other similar tools. Second, we provide a discussion of classroom experiences. Third we provide results from a small experimental study introducing Frances to groups of randomly selected students with various levels of computer architecture experience.

5.1. Tool Comparison

First, we compare the retrieval and setup of various systems in Table 1. This includes both systems that help with understanding program control flow, code generation, and assembly language as well as those that help illustrate architectural details.

This table shows that while many of the tools work on a variety of systems, there are minor limitations to where they can be used. Our tool may be used on any system that has a web browser. Further, while there are a couple systems that only require downloading a Java jar file to run (a JVM must be installed), ours is the only that requires no downloading or installation whatsoever. Finally, most of the tools we compare against are also free.

Next, in Table 2, we compare how various tools represent and interact with the user with respect to the low-level language.

This table shows that many of the related tools also use a real assembly language. However, unlike ours, most use only RISC languages and do not have the flexibility to change assembly languages. For comparing high-level and low-level languages we currently support Intel and AT&T x86 syntax and MIPS. Adding additional languages simply involves installing a cross compiler. Next, nearly all

Table 1. Comparison of Tool Setup

Tool	OS Platform	Req. Install	Free
Frances	Any	No	Yes
MarieSIM	JVM	Download Jar	Yes
Mieru Compiler	Unix variants,Windows	Build	Yes
GSPIM	Unknown	Unknown	Unknown
SimpleScalar (ss-vis)	Unix variants	Yes	Academic
Simple Computer	Source	Build	Yes
GDB	Unix variants,Windows	Yes	Yes
KDBG	Linux	Yes	Yes
ICD-C	Linux,Solaris,Windows	Yes	No
Avora	JVM	Download Jar	Yes
aiCall	Linux,Mac,Windows	Yes	Academic[3]

Table 2. Comparison of Tool Assembly Representation. Irrelevant tools from Table 1 removed.

Tool	Asm. Lang	HLL ↔ ASM	Instruction Ordering	CF Graph	Path Types	Coloring
Frances	Real	Yes	Actual	Yes	Intra-procedural	Yes
Mieru Compiler	Real	Text	Actual (text)	No	N/A	Current code
GSPIM	Real (MIPS)	No	Not-actual	Yes	Calls vs. Jumps	Current node
ICD-C	N/A	N/A	Not-actual	Yes	Unknown	Yes
Avora	Real (AVR)	N/A	Not-actual	Yes	Calls vs. Jumps	Entry points
aiCall	N/A	N/A	Not-actual	Yes	Yes	Decision nodes, Entry points, Edge types

the tools do not show how user entered high-level code corresponds to low-level code. Debuggers and debugger front-ends frequently show the version of the program side-by-side (users may also do this manually), but this is no more than a simple text comparison. Ours is the only tool we know of that automatically shows how the two versions compare and breaks the complex assembly code into its control structures graphically. Similar to text-based representations, which are difficult to understand for beginners, our tool maintains actual instruction ordering in our graphical representation of the machine code. While nearly all the tools in the table display a control flow graph, none of these graph based tools maintain actual instruction order. Next, most of the tools also distinguish between path types, however, most only distinguish between inter- vs intra-procedural edges (sometimes as separate graphs). Finally, most of the tools also use some sort of coloring to make reading the control flow graphs easier. However, most are quite limited in what they differentiate between. For tools that illustrate execution, they often just highlight the current node. Others often just show procedure entry points or decision nodes.

Now, in Table 3, we compare how various tools represent and interact with the user with respect to simulation and illustrating execution at the architecture level. Note that this is just a small subset of the tools that are more suitable in undergraduate level education. We leave out many research and advanced tools from this table (most of these are text/command line based).

Table 3. Comparison of Tool Machine Visualization. Irrelevant tools from Table 1 removed.

Tool	Simulation Language	Step Back	Graphical Machine	Color Machine	Edges Machine	Edges Color
Frances	HLLs, ASM	Yes	Yes	Yes	Yes	Yes
MarieSIM	ASM (Custom)	No	Yes	No	No	No
GSPIM	ASM (MIPS)	No	No	N/A	N/A	N/A
SimpleScalar (ss-vis)	Pisa Binary	No	Yes	Yes	No	N/A
Simple Computer	ASM (Custom)	No	No	N/A	N/A	N/A
GDB	HLLs,ASM	No	No	N/A	N/A	N/A
KDBG	HLLs,ASM	No	Yes	Yes	No	N/A

First, among the tools, there is quite a mix of assembly languages including real languages (both RISC and CISC) and custom languages. However, aside from the debuggers, ours is the only tool that also shows a high-level language version of the program being run and allows users to enter simulation code in a high-level language. Additionally, our tool is the only tool we found that allows users to step backwards through execution. We believe this to be an important feature for revisiting complicated steps or sets of steps. Finally, several of the tools create graphical version of the machine state like our tool. Some also even color code parts of the machine state (e.g. recently read or written values). However, ours is the only tool we found that illustrates pointers in the machine state and colors the edges of these pointers to show recent changes.

Summary In summary, this comparison illustrates the key feature differences between Frances and related tools. We show that Frances is easier to start using than related tools (platform independent, doesn't require install, and is free). Further, among related tools, Frances is the only tool that graphically shows relationships between high-level and low-level languages while maintaining instruction order. Finally, Frances is the only tool that allows students to step backwards to revisit complex steps/procedures and illustrates edges (pointers) in the machine state.

5.2. Classroom Experiences

Educational materials have been developed for use in several courses. The materials associated with the Frances tool have been successfully used in multiple sections of an upper level compiler construction course with a total of 20 students. The students enrolled in this course have had courses in data structures, computer architecture and operating systems. In the computer architecture course textbook [Stallings 2010], students were provided an introduction to instruction sets. However, the students have had no experience with assembly language programming. Frances was used in several class sessions to introduce assembly language. Then as various language constructs arose, Frances was used to investigate the low-level implementation of these constructs. Students reported that they found the tool easy to use and scored an average of 93.8% on the seven lab assignments using Frances to understand AT&T x86 assembly language. Furthermore, the students used the tool for reference in the development of a compiler created as part of the course. Students reported using the tool in an exploratory manner to gain insight into generating assembly code for the compiler constructed as a course project. Subsequently the Frances tool and course materials have been integrated into the compiler course curriculum at McKendree University.

Additionally, Frances and associated course materials have been introduced into a computer architecture course. The course materials, developed for integration into the course, allow the instructor to continue using a standard textbook in the subject and interject the Frances materials for various topics. Initially it was used to introduce the concept of translating from a high-level to a low-level language. Then it was used as an example of a real system for topics such as variable types, operands, addressing modes, and registers. The Frances tool has also been introduced as an explanatory tool in CS1 course offerings with promising results. During introductory discussions about the compilation process students were able to get a sense of the translation process from high-level to low-level language. Students gave positive comments in regards to understanding how instructions are processed in the computer. As part of our future work we plan to incorporate high-level language aspects into the Frances system in an effort to help facilitate further integration into CS1/CS2.

5.3. Empirical Study

Our classroom experiences with Frances indicated that the tool is useful for its intended purpose in providing students with a connection between low-level and high-level language translation, allowing students to gain experience with assembly languages and an insight into computer architecture in an easy to understand integrated environment. In an effort to collect more data regarding the effective usage of the Frances tool, an experimental design was developed. The goal of this initial study was to collect data for ease of use; usefulness in regards to making a connection between high-Level languages and low-Level languages; in understanding machine states; and in introducing assembly language. Twenty-nine students were randomly selected from undergraduates majoring in computer

science, computational science, and information systems. All of the students had completed a two course sequence in computer programming and some were currently enrolled in a data structures course. Since Frances focuses on making a connection between high-level and low-level languages, and computer architecture we stratified the group on computer architecture course experience. Some students had completed a course in computer architecture (DONE, N=4), some were currently enrolled (IP, N=12) and some had no experience with the subject (NONE, N=13). This course is a standard course in computer architecture using the textbook by Stallings [Stallings 2010].

5.3.1. Experimental Setup. The goals of this study are to investigate if students can easily begin using the Frances tool and how effective the tool is in conveying its intended information. As indicated from the classroom experience section of this paper, in the context of a course, students easily understood how to use the system and found it to be very useful component of the the course. Students using the system without the context of a course or instructor help provide a stronger indication of the tool's effectiveness and ease of use.

The students were gathered in a computer lab and given Lesson 1 (available at `http://www.cs.iastate.edu/~sapha/tools/frances/frances-a/lessons/`). Three minutes of instruction on what tasks to complete followed. No instruction or background was given about the Frances system itself or the topics involved. Students were required to complete the Lesson 1 worksheet while an instructor was available to answer any questions, and then complete the Lesson 2 worksheet, found on the Frances website, on their own. Students had no prior knowledge of the system, its purpose, or the material covered in the system. The questions from these two lessons are shown in Table 4 and Table 5.

Table 4. Lesson 1 questions. These questions are preceded by a brief overview of how to access the tool, the interface, and how to read assembly instructions. Italicized text is not in the actual lesson.

Q#	Summary
1	Add one line to the default program that assigns "5" to an integer variable.
1a	What assembly code is the same as the default program?
1b	What assembly code is different?
1c	*A brief description of the code before and after the new code is elided.* Step through the program. Describe what is occurring in each of the new assembly lines.
2	Put each of the register shown in the machine state into the following categories (some may go into multiple categories): General Purpose, Data, Address, Condition Codes.
3	Given the register values in the following sample machine state *(the state from the right side of Figure 3).*
3a	Near which address in memory would one find the assembly code for the program being executed?
3b	After the next instruction executes, which register(s) will change value?
3c	After the next instruction executes, what will be the value of the changed register(s)?
4	Given the following example state. *Shown here as Figure 12.*
4a	At which memory location is the variable `x` stored? In hexadecimal? Relative to `ebp`?
4b	At which assembly line is the value 2 put into this memory location?
4c	Write the above C program in Frances. After execution of step 13, what are the values of `eax` and the stack? After step 14 what are these values?

5.3.2. Ease of Use. Students completed Lesson 1 in 45 to 53 minutes. The instructor only answered four student questions. One student did not understand to step through the instructions to see changes in the machine state. Two students did not understand which program was the default program. These students had no experience with computer architecture and required a brief explanation that it was assembly language and registers being displayed. One student wanted to know what each assembly instruction did and was instructed to try to figure it out from context and stepping forward and backward.

Students were able to read through Lesson 1, which provides a one page explanation of the Frances tool, including the connection between the source code and assembly code; the connection between assembly code and machine states; the color coding of assembly instructions; and the color scheme of the machine states. Additionally this page describes how to enter, compile and step

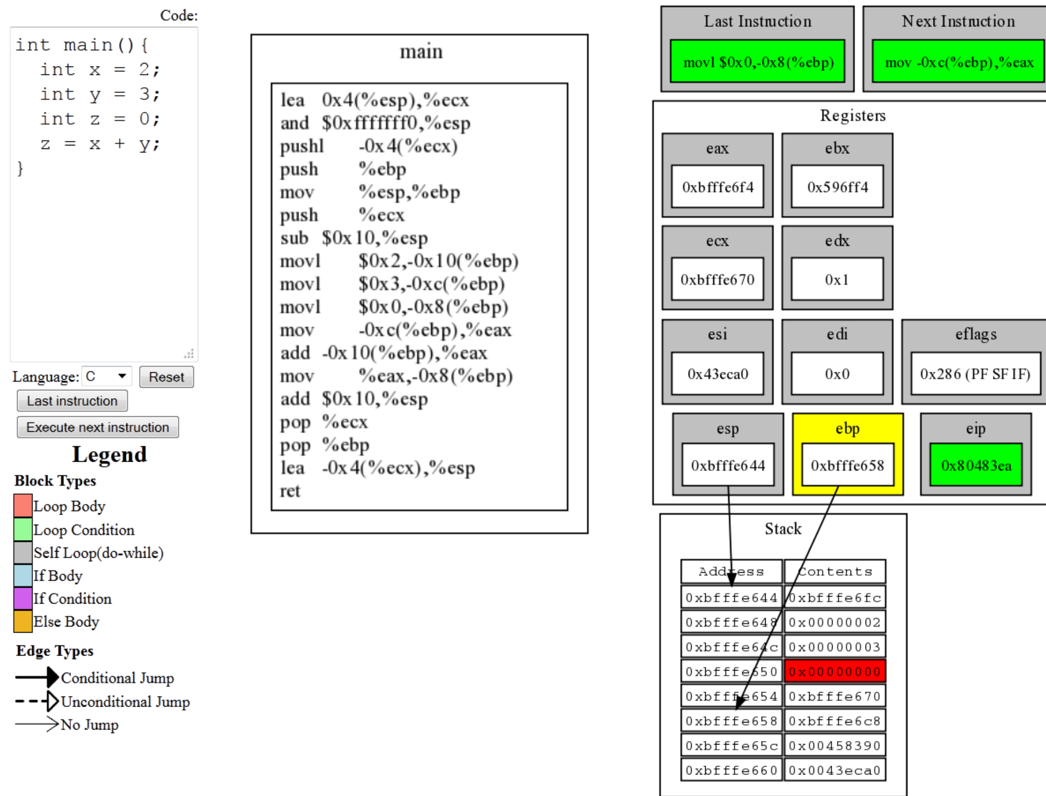

Fig. 12. Lesson 1 question 4 example.

Table 5. Lesson 2 questions. Italicized text is not in the actual lesson.

Q#	Summary
1	Go to the Frances-A site and type the following code into Frances-A. *Sample program shown with* `main` *only containing the statements:* `int n=5; char c='5'; float y=5;`
1a	Write the assembly code that stores 5 into `n`.
1b	Write the assembly code that stores '5' into `c`.
1c	Write the assembly code that stores 5 into `y`.
1c.i	Why do you think the float is stored differently than the int and char?
1c.ii	This is stored in the IEEE 754 standard. Convert it from this representation to its decimal value. Show all work.
2	Step through the program.
2a	How many bytes does the integer occupy?
2a.i	What is the address of and value stored in the memory location.
2b	How many bytes does the char occupy?
2b.i	What is the address of and value stored in the memory location.
2b.ii	What was stored in this address prior to execution of the instruction storing the char?
2c	How many bytes does the float occupy?
2c.i	What is the address of and value stored in the memory location.
3	What are the `mov`, `movb`, and `movl` instructions? Why are they different?
4	What is the purpose of the `sub` and `add` instructions that occur before and after these `mov` instructions?
5	Replace the line char c; with the line `char c[5] = {'1', '2', '3', '4', '5'};`
5a	Step through the program.
5b	How is this array stored? Write the addresses and value in the memory locations prior to moving any data and then after each `mov` associated with the array

through programs; a description of the assembly language format; and a description of registers, addresses, and value prefixes in the assembly instructions. These explanations are very brief, fitting on a single page.

Students then went on to use the Frances system completing the lessons and answering 25 questions pertaining to assembly language, computer architecture concepts, and machine states with an overall mean score of 91.25 out of 125. This equates to 75.70%. Thus, in general, *students were easily able to use the system with little instruction or context and successfully demonstrate an understanding of the material*. Stratifying these results across prior computer architecture course experience reveals more information. Students with no current or previous enrollment in a computer architecture course (NONE) had a mean score 70.30%, students currently enrolled in a computer architecture course (IP) had a mean score of 79.64%, and students who have completed a computer architecture course (DONE) had a mean score of 81.00%. Boxplots of this data are shown in Figure 13.

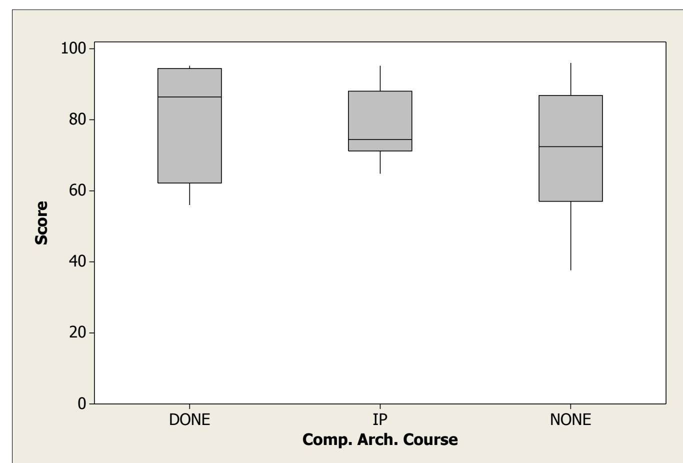

Fig. 13. Overall scores for lessons. DONE: Completed computer architecture course. IP: Currently enrolled in a computer architecture course. NONE: Not completed or enrolled in a computer architecture course.

When surveyed on a scale of 1-5 (1 being very difficult, 3 neutral and 5 very easy) students rated ease of use with a mean of 3.59 overall. Again, stratifying on computer architecture course experience NONE rated ease of use at 3.62, IP at 3.33 and DONE at 4.25. Box plots of this data are presented in Figure 14. This may indicate that the opinion of the ease of use is tied to an understanding of the material covered. Students with less context of the experience felt that the system was not as easy to use as those that had a completed a course covering the topics. However, more data would need to be gathered from students that have completed a computer architecture course (i.e., DONE).

In summary, students with no experience in the topics covered in these lessons were able to learn the tool on their own, with little to no instructor help, and complete the lessons which demonstrates that the tool is easy to learn to use.

5.3.3. Usefulness. In this subsection, we consider the usefulness of the tool. Table 6 provides the mean and standard deviation of the student responses to the lesson questions. The first three columns show the results for the respective computer architecture experience and the last column

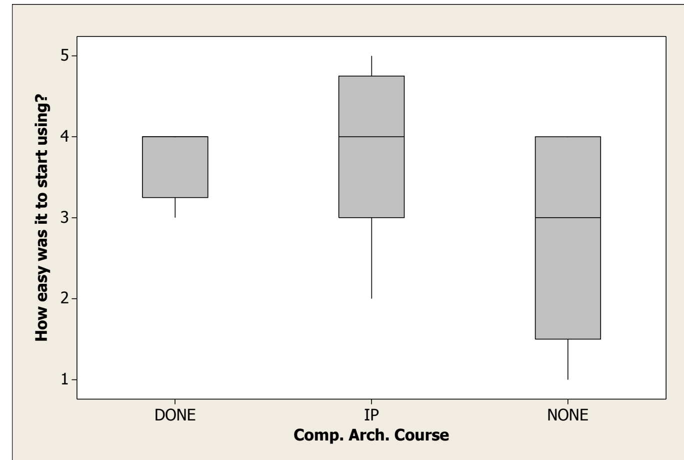

Fig. 14. Survey results on the opinion of ease of use.

provides the aggregate results for all students. The first row gives the results for all questions and the last three rows give the results for high-level/low-level, machine state, and assembly language concepts, respectively. Overall students were very successful with the system. As shown in the

Table 6. Scores for lessons 1 and 2 (%)

	NONE		IP		DONE		ALL	
	Mean	St.Dev.	Mean	St.Dev.	Mean	St.Dev.	Mean	St.Dev.
Overall	70.33	18.92	79.64	10.57	81.00	17.77	75.70	15.97
HL/LL	74.20	29.60	69.50	32.70	84.00	15.70	76.00	26.14
Machine St.	73.40	27.00	78.20	26.50	83.50	23.90	76.80	25.70
Assembly	87.50	24.17	91.82	16.62	82.50	35.00	88.52	22.48

first row of Table 6, the average score is 75.70%. Separating students by experience reveals more. Students with no background in computer architecture scored, on average, 70.33%. This indicates that in less than two hours, students without experience in the subject or proper context of the system were able to learn the system, use it and answer the questions with a 70.33% score. Students with experience in computer architecture scored 79.64% and 81.00% for IP and DONE, respectively. These students have experience with a traditional course in computer architecture, but no exposure to a real assembly language or architecture. Again they were able to successfully understand the system and material outside the context of a course, in less than two hours.

Next we break the data down into components to investigate the effectiveness of the system for aiding in the understanding of the connection between high-level and low-level Language, machine state constructs, and Assembly Language.

High-level and Low-level Language Relationships Students scored an average of 76.00% (NONE 74.20%, IP 69.50%, DONE 84.00%) with questions regarding the connection between high-level and low-level languages. The questions used for this assessment are 1c and 4a from Lesson 1 (Table 4), as well as 1a–c from Lesson 2 (Table 5). None of the students had previous experience with a real assembly language. The questions asked students to write a high-level language program, step through the resulting assembly language program segment and describe the result of each assembly instruction, relating it to the high-level code; identify the storage location of a variable by

relating the high-level instructions to the assembly instructions; and determine the function of other low-level instructions. All are short answer questions requiring students to use only the system in determining the answers with no other hints.

Machine State Students scored an average of 76.8% (NONE 73.4%, IP78.2%, DONE 83.5%) with questions about the machine state. Students were asked to step through program segments, indicate how register and stack values change; how and where various variable types are stored; and the process of storing values. Theses are the questions 3c, 4c from Lesson 1 (Table 4), and 2a, 2b, 2c, and 5 from Lesson 2 (Table 5). Again, all questions required the students to provide short answers without any outside resources.

Assembly Language Students scored an average of 88.52% (NONE 87.50%, IP 91.82%, DONE 82.50%) with questions directly addressing assembly language. These are questions regarding the format and function of several assembly instructions given in the assignment (3 and 4 from Lesson 2 (Table 5)). The first page of Lesson 1 provides a short description of x86 AT&T assembly language format. The results here are much higher than the other two concepts. The questions regarding assembly language were near the end of the Lesson 2 worksheet. For students to get to the end of this worksheet they had spent about one and a half hours working with the system. Also, the questions associated with high-level/low-level constructs and machine state involves some knowledge of assembly language. Thus, when these final questions are encountered the students have enough understanding of the subject to correctly answer questions.

This initial study provides evidence indicating the ease at which students can begin using the tool to introduce these computer architecture topics to students in the computing disciplines. Outside the context of a course and with no instruction, students with no background in computer architecture were able to start using the tool and answer questions about these topics scoring 70.33%. Students with course experience and thus some context of the topics, scored even higher. Stratifying on computer architecture course background is helpful. However, given the small sample size we consider the aggregate data. The overall results indicate that students can quickly begin using the system and successfully complete associated exercises.

6. CONCLUSION AND FUTURE WORK

Low-level details of computing are an important part of computer science curricula impacting core courses such as computer organization, compiler design, and programming languages [CC 2008]. Unfortunately, learning these topics is difficult since they are very different from the introductory high-level language programming courses that most students begin with [Loui 1988]. To ease the process of learning these low-level details, and help bridge the gap from the familiar high-level language concepts, we present Frances. The Frances tool includes: illustrating key differences between high-level and low-level languages, logical separation of components in machine states (including different register types), edge drawing to show control flow paths, pointer targets, and stack behavior, color coding to show the purpose of code as well as accesses and writes of components in the system to show behavior of each instruction, and finally the ability to step both backwards and forwards though execution.

Our preliminary studies show promise for the Frances tool. We have presented experimentation and classroom experiences that demonstrate both the ease of use and usefulness of these tools. Most importantly, students with no experience of computer architectures or low-level languages were able to complete assignments outside the context of a course and without instructor help. As part of our future work we intend to perform more detailed experimental studies.

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 high-level and low-level languages. Frances is designed to be easy to swap in different architectures, it simply requires installing additional software on the server and minor changes to the presentation. Third, we plan to extend the

tool to handle tracking values of high-level variables. Finally, we plan to continue evaluating and improving the tool based on both student and expert evaluations.

ACKNOWLEDGMENTS

The authors would like to thank the students who tested the tool as well as the anonymous reviewers for their comments and suggestions. Sondag and Rajan were supported in part by US NSF under grants 06-27354, 07-09217, 08-46059, and 11-17936.

REFERENCES

ABSINT. *aiSee - Graph Visualization*. http://www.absint.com/aisee/.

AIKEN, A. 1996. Cool: a portable project for teaching compiler construction. *SIGPLAN Not. 31,* 7, 19–24.

ALLEN, F. E. 1970. Control flow analysis. In *Symposium on Compiler optimization*. 1–19.

BINKERT, N. L., HALLNOR, E. G., AND REINHARDT, S. K. 2003. Network-oriented full-system simulation using M5. In *Workshop on Computer Architecture Evaluation using Commercial Workloads*.

BORUNDA, P., BREWER, C., AND ERTEN, C. 2006. GSPIM: graphical visualization tool for MIPS assembly programming and simulation. *SIGCSE Bull. 38,* 1, 244–248.

BRAUGHT, G. AND REED, D. 2001. The knob & switch computer: A computer architecture simulator for introductory computer science. *J. Educ. Resour. Comput. 1,* 4, 31–45.

BREDLAU, C. AND DEREMER, D. 2001. Assembly language through the Java virtual machine. In *SIGCSE '01: Proceedings of the 32nd ACM technical symposium on Computer science education*.

CC. 2008. Computing curricula 2008: An interim revision of cs 2001. *CC*. http://www.acm.org/education/curricula/ ComputerScience2008.pdf.

COE, P. S., HOWELL, F. W., IBBETT, R. N., MCNAB, R., AND WILLIAMS, L. M. 1995. An integrated learning support environment for computer architecture. In *WCAE-3 '97: Proceedings of the 1997 workshop on Computer architecture education*. ACM, New York, NY, USA, 8.

CONWAY, M. J. AND PAUSCH, R. 1997. Alice: easy to learn interactive 3d graphics. *SIGGRAPH Comput. Graph. 31,* 3, 58–59.

CORLISS, M. L. AND LEWIS, E. C. 2008. Bantam: a customizable, Java-based, classroom compiler. In *SIGCSE '08: Proceedings of the 39th ACM technical symposium on Computer science education*.

ELLSON, J., GANSNER, E. R., KOUTSOFIOS, E., NORTH, S. C., AND WOODHULL, G. 2001. Graphviz - open source graph drawing tools. *Graph Drawing*, 483–484.

ESPONDA-ARGUERO, M. 2008. Algorithmic animation in education—review of academic experience. *Journal of Educational Computing Research 39*, 1–15.

FREE SOFTWARE FOUNDATION. 2009. GNU BinUtils: a collection of binary tools. http://www.gnu.org/software/binutils/.

FREE SOFTWARE FOUNDATION. 2010. GDB: The GNU Project Debugger. http://www.gnu.org/software/gdb/.

GONDOW, K., FUKUYASU, N., AND ARAHORI, Y. 2010. Mierucompiler: integrated visualization tool with "horizontal slicing" for educational compilers. In *SIGCSE '10: Proceedings of the 41st ACM technical symposium on Computer science education*. ACM, New York, NY, USA, 7–11.

GRAHAM, N. 1985. *Introduction to computer science (3rd ed.)*. West Publishing Co., St. Paul, MN, USA.

HUNDHAUSEN, C. D. 2002. Integrating algorithm visualization technology into an undergraduate algorithms course: ethnographic studies of a social constructivist approach. *Comput. Educ. 39,* 3, 237–260.

KISE, K., KATAGIRI, T., HONDA, H., AND YUBA, T. 2004. The simcore/alpha functional simulator. In *WCAE '04: Proceedings of the 2004 workshop on Computer architecture education*. ACM, New York, NY, USA, 24.

LOUI, M. C. 1988. The case for assembly language programming. *IEEE Transactions on Education E-31,* 3, 160–164.

LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the conference on Programming language design and implementation*. 190–200.

MCNALLY, M., NAPS, T. L., FURCY, D., GRISSOM, S., AND TREFFTZ, C. 2007. Supporting the rapid development of pedagogically effective algorithm visualizations. *Journal of Computing Sciences in Colleges 23,* 1 (10/2007), 80–90.

MUCHNICK, S. S. 1997. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

NAPS, T. L., RÖSSLING, G., ALMSTRUM, V., DANN, W., FLEISCHER, R., HUNDHAUSEN, C., KORHONEN, A., MALMI, L., MCNALLY, M., RODGER, S., AND VELÁZQUEZ-ITURBIDE, J. A. 2002. Exploring the role of visualization and engagement in computer science education. In *ITiCSE-WGR '02: Working group reports from ITiCSE on Innovation and technology in computer science education*. ACM, New York, NY, USA, 131–152.

NIKOLIC, B., RADIVOJEVIC, Z., DJORDJEVIC, J., AND MILUTINOVIC, V. 2009. A survey and evaluation of simulators suitable for teaching courses in computer architecture and organization. *IEEE Transactions on Education 52*, 449 – 458.

NULL, L. AND LOBUR, J. 2003. MarieSim: The MARIE computer simulator. *J. Educ. Resour. Comput. 3,* 2, 1.

PAI, V. S., RANGANATHAN, P., AND ADVE, S. V. 1997. RSIM: Rice simulator for ILP multiprocessors. *SIGARCH Comput. Archit. News 25,* 5, 1.

PATT, Y. N. AND PATEL, S. J. 2004. *Introduction to Computing Systems: From Bits & Gates to C & Beyond*, 2 ed. McGraw-Hill, Inc., New York, NY, USA.

PAUW, W. D., JENSEN, E., MITCHELL, N., SEVITSKY, G., VLISSIDES, J. M., AND YANG, J. 2002. Visualizing the execution of Java programs. In *Revised Lectures on Software Visualization, International Seminar*. Springer-Verlag, London, UK, 151–162.

POWERS, K., GROSS, P., COOPER, S., MCNALLY, M., GOLDMAN, K. J., PROULX, V., AND CARLISLE, M. 2006. Tools for teaching introductory programming: what works? *SIGCSE Bull. 38,* 1, 560–561.

PRICE, B. A., SMALL, I. S., AND BAECKER, R. M. 1992. A taxonomy of software visualization. *Journal of Visual Languages and Computing 4*, 211–266.

REISS, S. P. 2003. Visualizing Java in action. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*. ACM, New York, NY, USA, 57–ff.

RESLER, R. D. AND DEAVER, D. M. 1998. Vcoco: a visualisation tool for teaching compilers. *SIGCSE Bull. 30,* 3, 199–202.

RÖSSLING, G. AND VELÁZQUEZ-ITURBIDE, J. Á. 2009. Editorial: Program and algorithm visualization in education. *Trans. Comput. Educ. 9,* 2, 1–6.

SANDERS, D. AND DORN, B. 2003. Jeroo: a tool for introducing object-oriented programming. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*. ACM, New York, NY, USA, 201–204.

SCHWETMAN, H. 1986. Csim: a c-based process-oriented simulation language. In *Conference on Winter simulation*. ACM, New York, NY, USA, 387–396.

SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2002. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. ACM, New York, NY, USA, 45–57.

SIXT, J. 2010. A graphical debugger interface. http://www.kdbg.org/.

SONDAG, T. AND RAJAN, H. 2010. A more precise abstract domain for multi-level caches for tighter wcet analysis. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*.

SONDAG, T. AND RAJAN, H. 2011. Phase-based tuning for better utilization of performance-asymmetric multicore processors. In *CGO: Proceedings of the 9th annual IEEE/ACM International Symposium on Code Generation and Optimization*.

STALLINGS, W. 2010. *Computer Organization and Architecture: Designing fro Performance (8th ed.)*. Pearson Education Inc., Upper Saddle River, New Jersy, USA.

STONE, J. A. 2006. Using a machine language simulator to teach CS1 concepts. *SIGCSE Bull. 38,* 4, 43–45.

SUNDARARAMAN, J. AND BACK, G. 2008. HDPV: interactive, faithful, in-vivo runtime state visualization for C/C++ and Java. In *SoftVis '08: Proceedings of the 4th ACM symposium on Software visualization*. ACM, New York, NY, USA, 47–56.

THE INFORMATIK CENTRUM DORTMUND (ICD). *ICD-C Compiler Framework*. http://www.icd.de/es/icd-c/.

TITZER, B., LEE, D., AND PALSBERG, J. 2005. Avrora: Scalable sensor network simulation with precise timing. In *Information Processing in Sensor Networks (IPSN)*.

URQUIZA-FUENTES, J. AND VELÁZQUEZ-ITURBIDE, J. Á. 2009. A survey of successful evaluations of program visualization and algorithm animation systems. *Trans. Comput. Educ. 9,* 2 (06/2009), 1–21.

WOLZ, U., MALONEY, J., AND PULIMOOD, S. M. 2008. 'scratch' your way to introductory cs. In *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*. ACM, New York, NY, USA, 298–299.

ZENG, H., YOURST, M., GHOSE, K., AND PONOMAREV, D. 2009. MPTLsim: a cycle-accurate, full-system simulator for x86-64 multicore architectures with coherent caches. *SIGARCH Comput. Archit. News 37,* 2, 2–9.

ZILLES, C. 2005. Spimbot: an engaging, problem-based approach to teaching assembly language programming. *SIGCSE Bull. 37*, 106–110.

Received R; revised e; accepted c
eived Month Year; Month Year; accepted Month Year