An Extension of Ripes RISC-V Simulator

Daichi Higashi and Hitoshi Oi[1,2] (Email: {s1300040,oi}@oslab.biz)

[1]School of Computer Science & Engineering, The University of Aizu, JAPAN. [2]Authors are order alphabetically.

会津大学

Introduction

The book by Hennessy and Patterson has been used for the undergrad computer architecture courses at the schools around the world for many decades [2]. It is originally based on the MIPS instruction set architecture (ISA) together with the MIPS ISA simulator, spim. However, the same book was re-written with the RISC-V ISA in 2017 and now the 2nd edition is available.

Ripes is one of the graphical processor simulators for the RISC-V ISA [3]. It visualizes the interactions between the microarchitectural components for the execution of each RISC-V instruction in the program. It has processor models that follow the implementation and improvement steps in [2]; from the single cycle (non-pipelined) model to the dual-issue 6-stage pipeline model. Also, the RISC-V toolchain is integrated into the simulator so that the C-language source code can be compiled and executed within the simulator environment.

We are currently extending the functionality of Ripes under the Student Cooperative Class Project (SCCP) at the University of Aizu.[a] In particular, we have added the following features to Ripes: (1) visualization of the calling stack frame and (2) fast forward button in the C-language line step. Figure 1 shows the screenshot of the Ripes simulator with the extended features (highlighted with blue lines and numbers (I) to (IV)). We describe these extended features in the following sections.

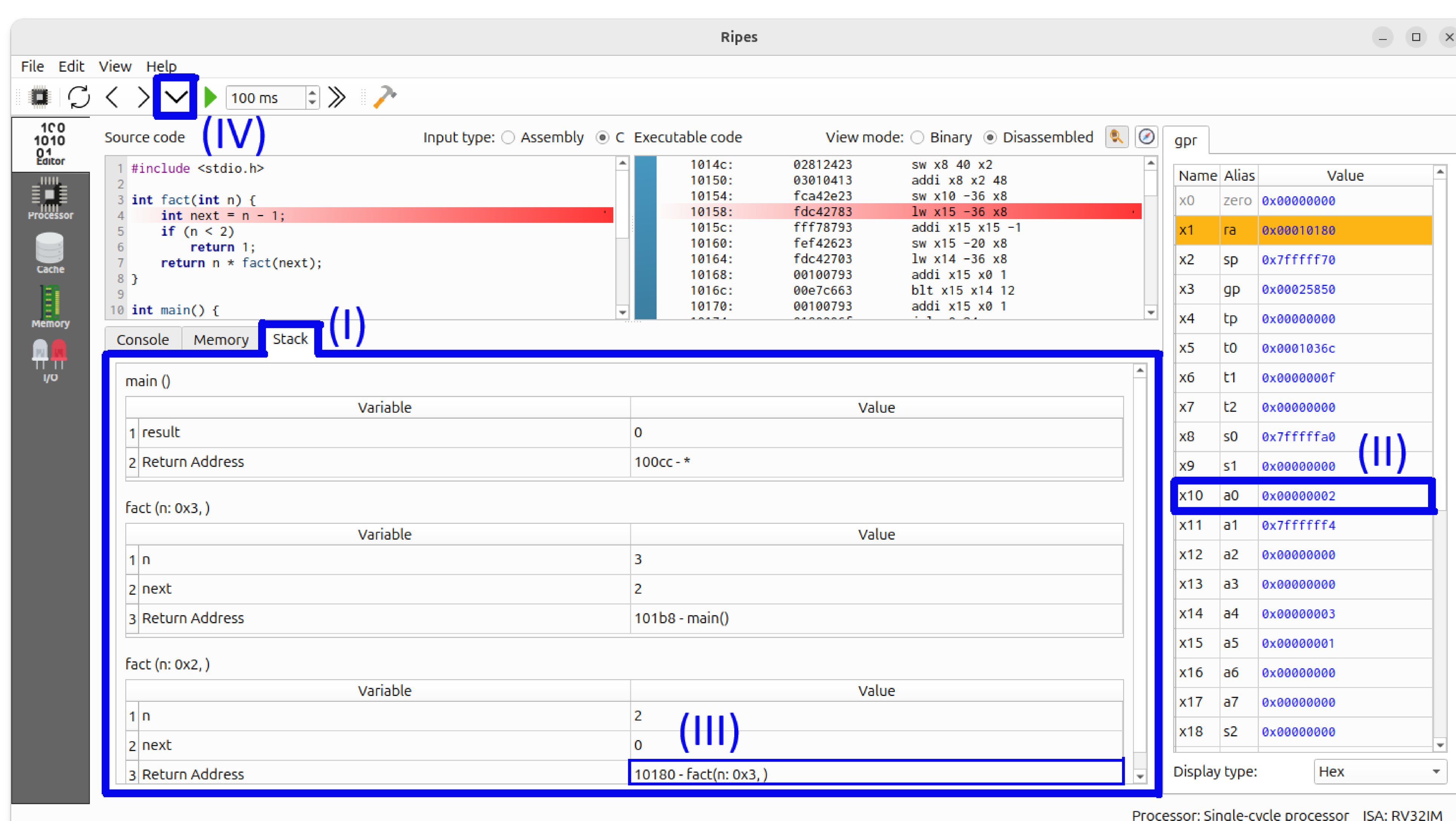

Figure 1: Ripes with extended features.

[a]Something corresponding to the Directed Individual Study in the US universities.

Calling Stack Frame

The first example of the program code for learning how a processor executes machine instructions may be of few lines long and flat one. A further step toward a little more realistic programs can be those using function calls: consolidate common operations into a function and call it from the `main` function. When a function call occurs, a stack frame is created for storing function arguments, local variables and return address from the function. We have added a tab to display the calling stack frame for each function call ((I) in Figure 1). Each row of the frame table is a pair of variable name and its value. The information of calling frame structure, local variable names and entry/exit points of each function is obtained with libdwarf [1].

As an example, we use a C program executing the factorial function which invokes itself recursively. Figure 1 shows Ripes with stack tab showing the stack frame after the `fact` function has been invoked 2 times recursively. There are three call frames: one for `main` and two for `fact` functions. Two call frames of `fact` can be distinguished with the value of n. These call frames are stacked until the end of recursion condition holds ($n < 2$), then will be removed on return from each invocation one by one. In the current design, calling frames are stacked downward following the direction of the stack space growth in the memory address space. We may change this design to more intuitive behavior of the stack growing upward direction.

Please note that the function argument for `fact`, n is passed by the x10 (a0) register. It is then copied to the local variable location within the call frame by the function prologue as shown in Figure 1-(II). The value of return address is associated with the caller's function name with its argument ((III) in Figure 1). For example, the return address of `fact(2)` frame is 0x10180 with `fact(3)` as the calling function.

C Source Code Line Step Fast-Forward

Ripes has execution control buttons, such as Clock (proceed a single clock cycle, > button), Reverse (undo Clock execution, < button) and Run (continue execution until hitting a breakpoint or system call invocation, ▷ button). We have added a button (∨) to continue execution until reaching to the first instruction of the next C-source code line, which we call C-step button, between Clock and Run buttons ((IV) in Figure 1). The same operation can be performed by setting a breakpoint and click the Run button. However, the C-step button should allow users to move to the instruction of the interest more quickly with fewer manual operations by the users.

Figure 2 shows the C-source code (line 4) and RISC-V assembly (address 10158) windows before clicking the C-step button. By clicking the C-step button, the C-source code window proceeds to the next line (line 5) and the RISC-V window moves to (and highlight) the corresponding instruction (address 10164) as shown in Figure 3.

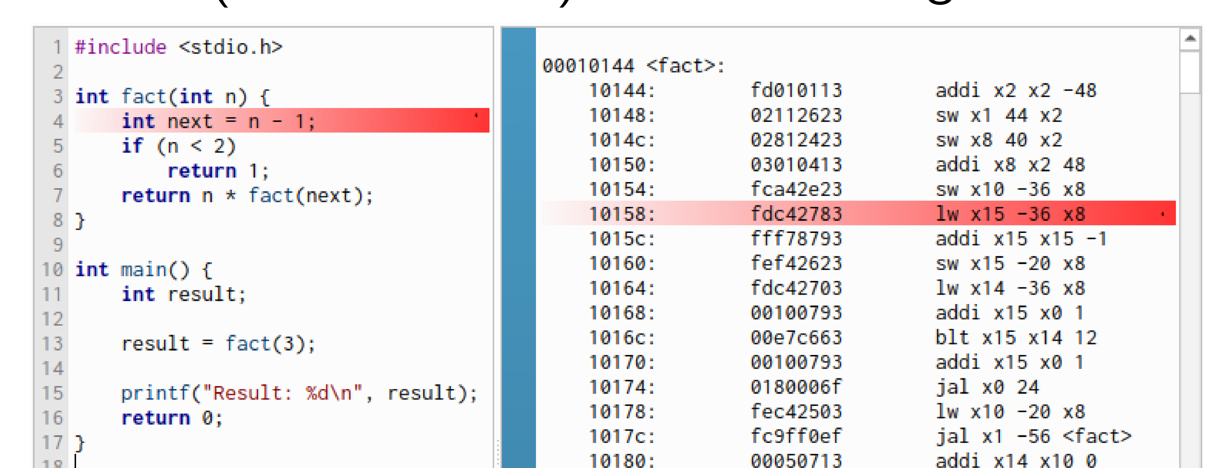

Figure 2: Before C-Step

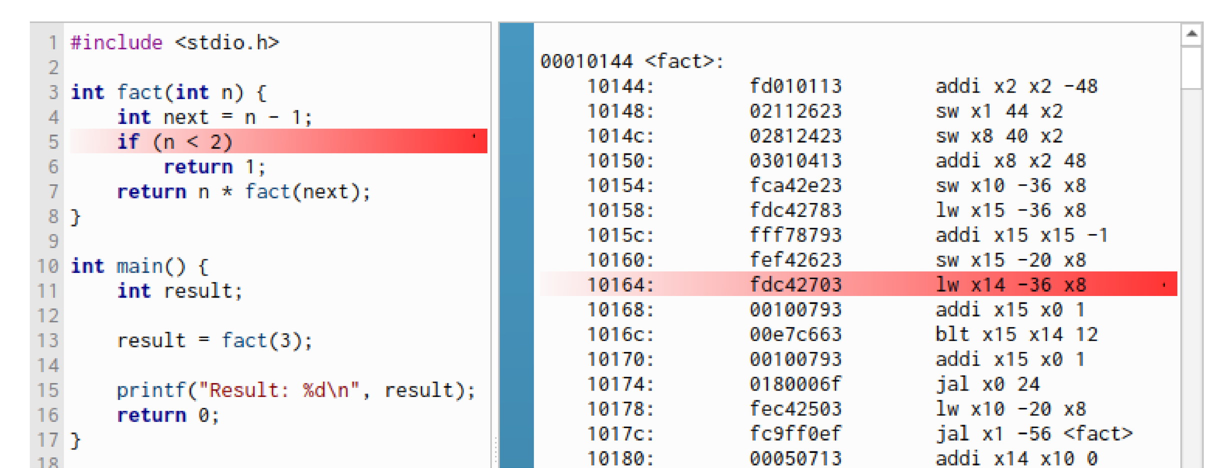

Figure 3: After C-Step

Summary and Future Work

We have extended the Ripes RISC-V simulator by adding the stack segment viewer and C-line step button. These features should assist the early learners of the computer architecture with focus the on hardware/software interface, especially for those who study the subject with the RISC-V based course materials.

For compiling the test C-programs, we have used the "-O0" option, so that the boundary of C statements is kept in the RISC-V instruction sequence. We have taken this option because this extension (and also Ripes) is targeted to those who are in the early stage of learning the computer architecture and related area. An alternative is to raise the optimization level and give different colors to the background of RISC-V instruction lines according to the C-program statements.

References

[1] David Anderson.
Dwarf page.
https://www.prevanders.net/dwarf.html.

[2] John L. Hennessy and David A. Patterson.
Computer Organization and Design RISC-V Edition.
Morgan Kaufmann Publishers Inc., 1st edition, 2017.

[3] Morten Borup Petersen.
Ripes.
https://github.com/mortbopet/Ripes.