Undergraduate Lab Report

Course Title: Experiment of Computer Organization

Course No: 60080014

Student Name: ______________________________

Student No: ______________________________

School: International School

Department: ______________________________

Major: Computer Science & Technology

Instructor： SUN Heng

Academic Year:202~202,Semester : 1st [√] 2nd[ ]

Academic Affairs Office of Jinan University

Date (dd/mm/yyyy)__________

Computer Organization Lab List

Student Name:________________ Student No:__________________

ID	Lab Name	Type
1	Number Storage Lab	Individual
2	Manipulating Bits	Individual
3	Simulating Y86-64 Program	Individual
4	Performance Lab	Team
5	A Simple Real-life Control System	Team
6	System I/O	Individual

Undergraduate Lab Report of Jinan University

Course Title Experiment of Computer Organization Evaluation______

Lab Name Simulating Y86-64 Program Instructor SUN Heng

Lab Address____

Student Name__________________________Student No______________

College International School

Department__________________________Major CST

Date_______/_______/_______ Afternoon

1. Introduction

In this lab, you will learn about the implementation of a Y86-64 processor. When completing this lab, you will have a keen appreciation for the interactions between instruction and hardware that affect your programs. You will run two simple Y86-64 programs and become familiar with the Y86-64 tools.

2. Lab Instructions or Steps

- Find Sources List:

Make sure the the link in **sources.list** match your ubuntu version:

https://wiki.ubuntu.org.cn/%E6%BA%90%E5%88%97%E8%A1%A8

- Backup: sources.list.bak

 # sudo cp /etc/apt/sources.list /etc/apt/sources.list.bak

- Modify: Rewrite sources.list using source from the website.

 # sudo gedit /etc/apt/sources.list

- Update:

 # sudo apt-get update (upgrade / -f install)

 # sudo apt-get install flex bison

For Ubuntu18、16、14:

 # sudo apt-get install tcl8.5 tcl8.5-dev tk8.5 tk8.5-dev

For Ubuntu18、16、14:

In **sim** and **seq** directory, find **Makefile**, and modify it:

```
TKLIBS=-L/usr/lib -ltk8.5 -ltcl8.5

# Modify the following line so that gcc
# header files on your system. Comment
# Tcl/Tk.

TKINC=-isystem /usr/include/tcl8.5
```

(1) Start by copying the **Lab3** to a directory in which you plan to do your work.

(2) Unpacked the file using:

tar xvf sim.tar (Or just double click and choose **extract**)

The directory sim contains the following subdirectories:

misc Source code files for utilities such as YAS (the Y86-64 assembler), and YIS (the Y86-64 instruction set simulator). It also contains the isa.c source file that is used byall of the processor simulators.

y86-code Y86-64 assembly code for many of the example programs. As a running example, we will use the program asum.ys in this subdirectory. The compiled version of the program is shown in Figure 1.

```
                            | # Execution begins at address 0
0x000:                      |   .pos 0
0x000: 30f40002000000000000 |   irmovq stack, %rsp      # Set up stack pointer
0x00a: 803800000000000000   |   call main               # Execute main program
0x013: 00                   |   halt                    # Terminate program
                            |
                            | # Array of 4 elements
0x018:                      |   .align 8
0x018: 0d000d000d000000     | array:    .quad 0x000d000d000d
0x020: c000c000c0000000     |   .quad 0x00c000c000c0
0x028: 000b000b000b0000     |   .quad 0x0b000b000b00
0x030: 00a000a000a00000     |   .quad 0xa000a000a000
                            |
0x038: 30f71800000000000000 | main:     irmovq array,%rdi
0x042: 30f60400000000000000 |   irmovq $4,%rsi
0x04c: 805600000000000000   |   call sum                # sum(array, 4)
0x055: 90                   |   ret
                            |
                            | # long sum(long *start, long count)
                            | # start in %rdi, count in %rsi
0x056: 30f80800000000000000 | sum:      irmovq $8,%r8         # Constant 8
0x060: 30f90100000000000000 |   irmovq $1,%r9         # Constant 1
0x06a: 6300                 |   xorq %rax,%rax        # sum = 0
0x06c: 6266                 |   andq %rsi,%rsi        # Set CC
0x06e: 708700000000000000   |   jmp     test          # Goto test
0x077: 50a70000000000000000 | loop:     mrmovq (%rdi),%r10   # Get *start
0x081: 60a0                 |   addq %r10,%rax        # Add to sum
0x083: 6087                 |   addq %r8,%rdi         # start++
0x085: 6196                 |   subq %r9,%rsi         # count--.  Set CC
0x087: 747700000000000000   | test:     jne    loop          # Stop when 0 #
0x090: 90                   |   ret                   # Return
                            |
                            | # Stack starts here and grows to lower addresses
0x200:                      |   .pos 0x200
0x200:                      | stack:
```

Figure 1 Sample object code file. This code is in the file asum.yo in the y86-code subdirectory.

(3) Change to the **sim** directory and build the Y86-64 tools:

 # make clean

 # make

(4) For the Y86-64 processor, the simulator can be run in TTY or GUI mode:

TTY mode Uses a minimalist, terminal-oriented interface. Prints everything on the terminal output. Not very convenient for debugging but can be installed on any system and can be used for automated testing. The default mode for all simulators.

GUI mode Has a graphic user interface, to be described shortly. Very helpful for visualizing the processor activity and for debugging modified versions of the design. Requires installation of Tcl/Tk on your system. Invoked with the -g command line option. Running in GUI mode is only possible from within the directory (seq) in which the executable simulator program is located.

For the simulator, you can specify several options from the command line:

-h Prints a summary of all of the command line options.

-g Run the simulator in GUI mode (the default is TTY mode).

-t (TTY mode only) Runs both the processor and the ISA simulators, comparing the resulting values of the memory, register file, and condition codes. If no discrepancies are found, it prints the message 'ISA CheckSucceeds.' Otherwise, it prints information about the words of the register file or memory that differ.

-l m (TTY mode only) Sets the instruction limit, executing at most m instructions before halting (the default limit is 10000 instructions).

-v n (TTY mode only) Sets the verbosity level to n, which must be between 0 and 2 with a default value of 2.

If you are running in GUI mode, you'll need to install Tcl/Tk along with the Tcl and Tk developer's packages.

(5) In **seq** directory, use following command to open GUI:

 # make clean

 # make ssim VERSION=std

 # ./ssim -g ../y86-code/asum.yo

This code runs SSIM in GUI mode, executing the instructions in object code file *asum.yo* from the y86-code subdirectory.

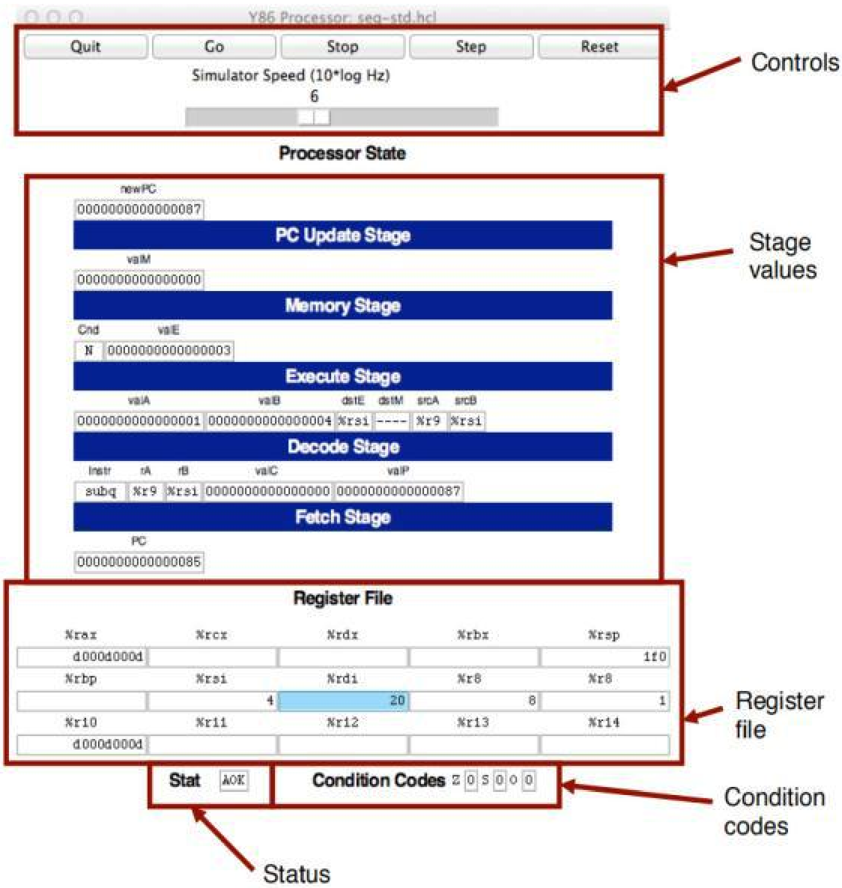


Figure 2 Main control panel for SEQ simulator

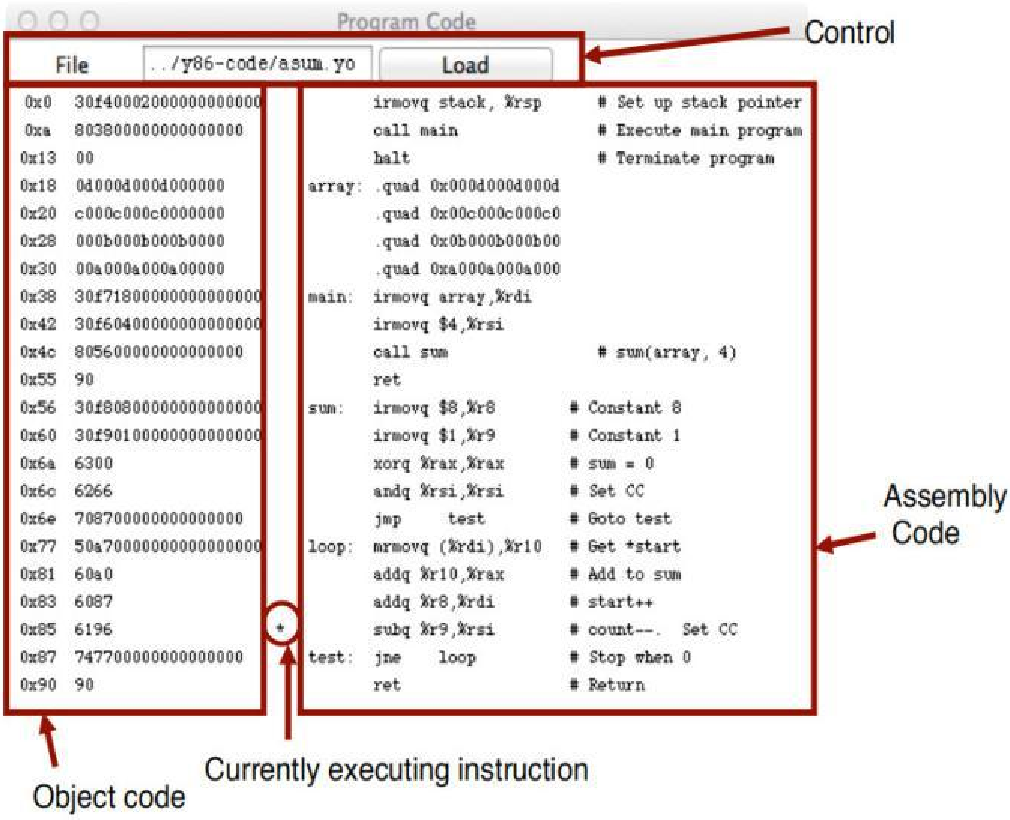


Figure 3 Code display window for SEQ simulator

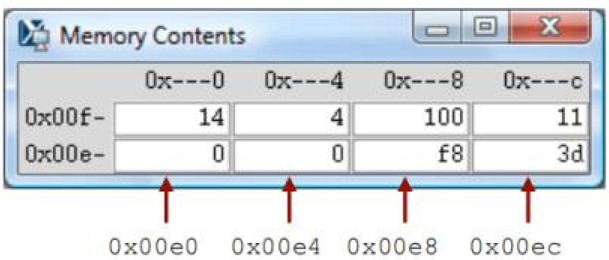


Figure 4 Memory display window for SEQ simulator

The main control window (Figure 2) contains buttons to control the simulator as well as status information about the state of the processor. The different parts of the window are labeled in the figure:

Control: The buttons along the top control the simulator. Clicking the *Quit* button causes the simulator to exit. Clicking the *Go* button causes the simulator to start running. Clicking the *Stop* button causes the simulator to stop temporarily. Clicking the *Step* button causes the simulator to execute one instruction and then stop. Clicking the *Reset* button causes the simulator to return to its initial state.

The slider below the buttons controls the speed of the simulator when it is running. Moving it to the right makes the simulator run faster.

Stage values: This part of the display shows the values of the different processor signals during the current instruction evaluation. The main difference is that the simulator displays the name of the instruction in a field labeled In str, rather than the numeric values of icode and ifun. Similarly, all register identifiers are shown using their names, rather than their numeric values, with "----" indicating that no register access is required.

Register file: This section displays the values of the 15 program registers. The register that has been updated most recently is shown highlighted in light blue. Register contents are not displayed until after the first time they are set to non-zero values.

Remember that when an instruction writes to a program register, the register file is not updated until the beginning of the next clock cycle. This means that you must step the simulator one more time to see the update take place.

Stat: This shows the status of the current instruction being executed. The possible values are:

AOK: No problem encountered.

ADR: An addressing error has occurred either trying to read an instruction or trying to read or write data. Addresses cannot exceed *0x0FFF*.

INS: An illegal instruction was encountered.

HLT: A *halt* instruction was encountered.

Condition codes: These show the values of the three condition codes: *ZF*, *SF*, and *OF*.

Remember that when an instruction changes the condition codes, the condition code register is not updated until the beginning of the next clock cycle. This means that you must step the simulator one more time to see the update take place.

Example:

The processor state illustrated in Figure 2 is for the first execution of line 29 of the *asum.yo* program shown in Figure 1. We can see that the program counter is at *0x085*, that it has processed the instruction *addq%r8*, *%rdi*, that register *%rax* holds *0xd000d000d*, the sum of the first array element, and *%rsi* holds 4, the count that is about to be decremented. Register *%rdi* holds *0x020*, the address of the second array element. There is a pending write of *0x03* to register *%rsi*(since *dstE* is set to *%rsi* and *valE* is set to *0x03*). This write will take place at the start of the next clock cycle.

The window depicted in Figure 3 shows the object code file that is being executed by the simulator. The editbox identifies the file name of the program being executed. You can edit the file name in this window and click the Load button to load a new program. The left-hand side of the display shows the object code being executed, while the right-hand side shows the text from the assembly code file. The center has an asterisk(*) to indicate which instruction is currently being simulated. This corresponds to line 29 of the *asum.yo* program shown in Figure 1.

The window shown in Figure 4 shows the contents of the memory. It shows only those locations between the minimum and maximum addresses that have changed since the program began executing. Each row shows the contents of two memory words. Thus, each row shows 16 bytes of the memory, where the addresses of the bytes differ in only their least significant hexadecimal digits. To the left of the memory values is the “root” address, where the least significant digit is shown as “-”. Each column then corresponds to words with least significant address digits *0x0*, and *0x8*. The example shown in Figure 4 has arrows indicating memory locations *0x01f0* and *0x01f8*.

The memory contents illustrated in the figure show the stack contents of the *asum.yo* program shown in Figure 1 during the execution of the *sum* procedure. Looking at the stack operations that have taken place so far, we see that *%rsp* was initialized to *0x200* (line 3). The call to *main* on line 4 pushes the return pointer *0x013*, which is written to address *0x01f8*. Procedure *main* calls *sum* (line 16), causing the return pointer *0x055* to be written to address *0x01f0*. That accounts for all of the words shown in this memory display, and for the stack pointer being set to *0x01f0*.

(6) You will be working in directory **sim/misc** in this part. Your task is to simulate the following two Y86-64 programs. The required behavior of these programs is defined by the example C functions in ***sim\misc\examples.c***.

(6.1) **sum.ys**: Iteratively sum linked list elements

Simulate a Y86-64 program **sum.ys** that iteratively sums the elements of a linked list. It consists of some code that sets up the stack structure, invokes a function, and then halts. In this case, the function should be Y86-64 code for a function (sum list) that is functionally equivalent to the C sum list function in Figure 1.

(6.2) **rsum.ys**: Recursively sum linked list elements

Simulate a Y86-64 program *rsum.ys* that recursively sums the elements of a linked list. It is similar to the code in *sum.ys*, except that it uses a function *rsum* list that recursively sums a list of numbers.

(7) Testing the program. The Y86-64 assembler takes a Y86-64 assembly

code file with extension **.ys** and generates a file with extension **.yo**. Put **rsum.ys** and **sum.ys** into **misc** directory. Use command:

 # make sum.yo

 # make rsum.yo

And then run the program in **seq** directory using:

 # ./ssim -g ../misc/sum.yo

 # ./ssim -g ../misc/rsum.yo

(8) For the programs **sum.ys** and **rsum.ys**, the result will be shown in register **%rax** with the sum of **0xcba**.

(9) Explain the function and processor state of each step in the source file of **sum.ys** and **rsum.ys** according to your snapshot.

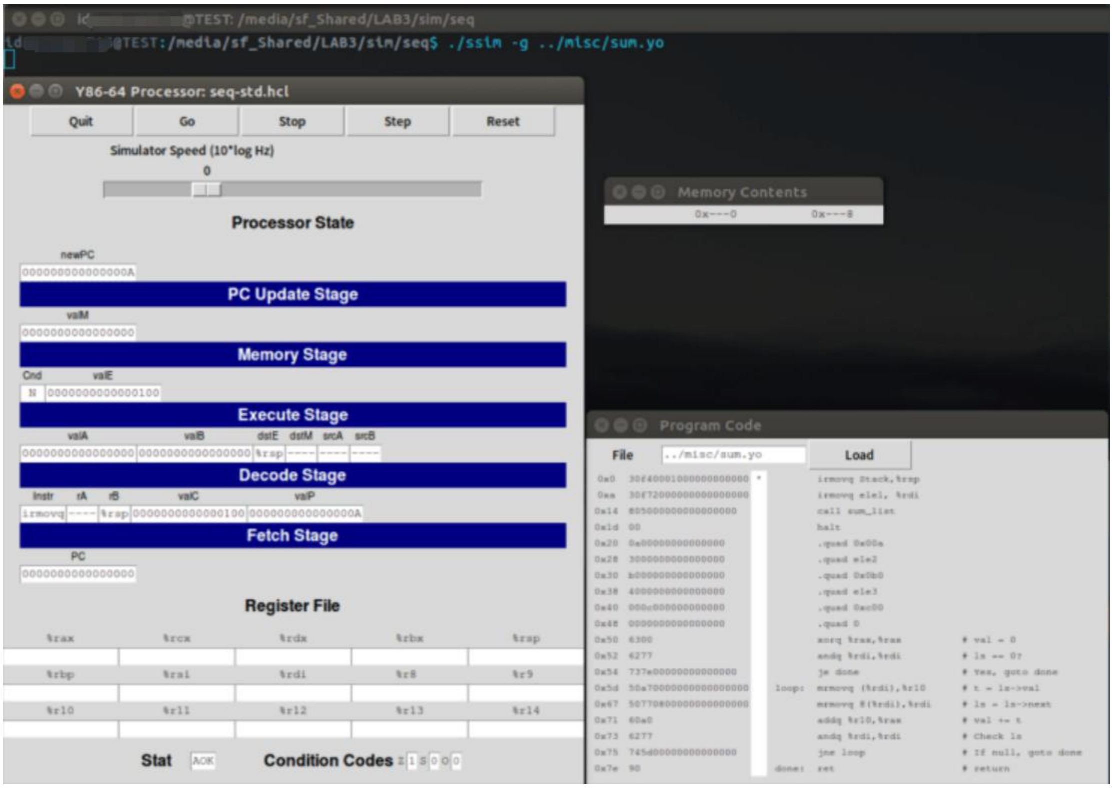
3. Lab Device or Environment

Ubuntu 16.04 (64-bit) with AMD Ryzen 9 5900HS CPU @ 3.30GHz and 4GB memory on virtual machine (Oracle VM VirtualBox)

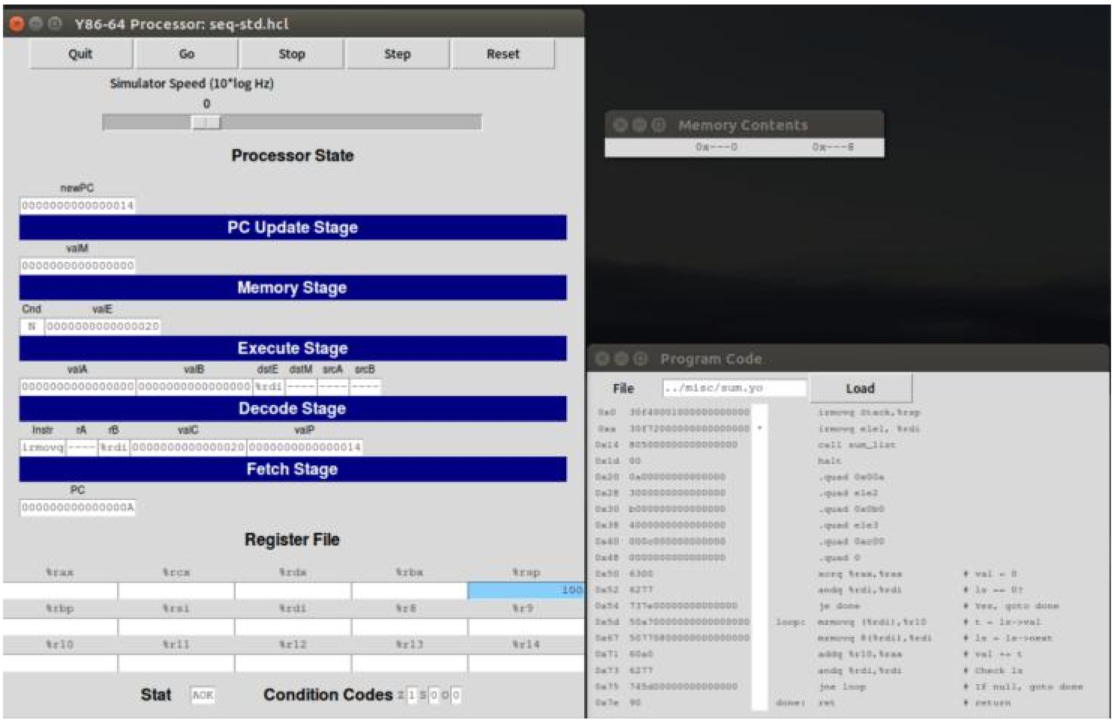
4. Results and Analysis

- Results and Analysis for **sum.yo**

(1) Initialize the stack pointer so that it points to 0x100.

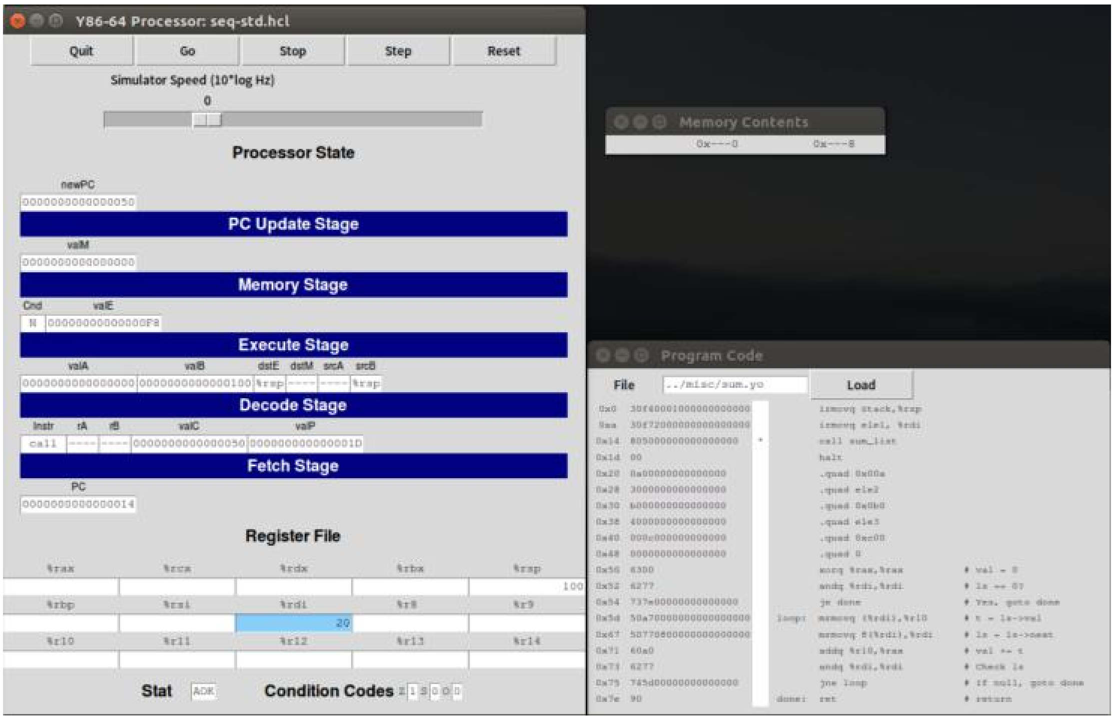


(2) Use IRMOVQ instruction to put the first node address of the linked list into the %rdi register.

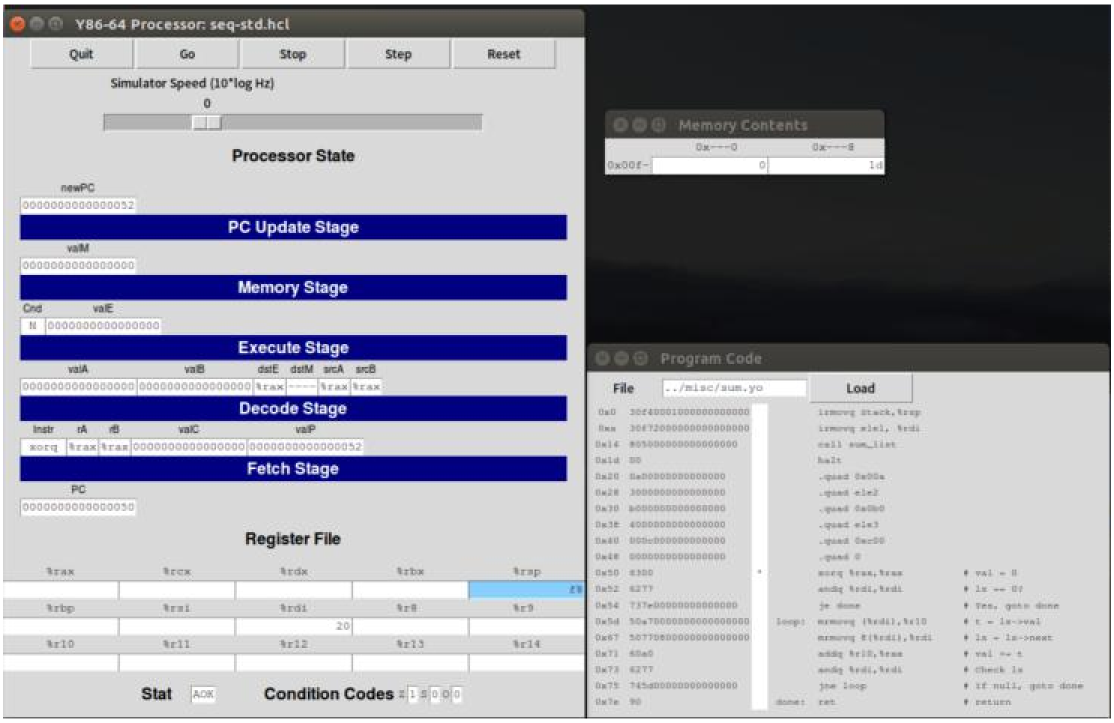


(3) The CALL instruction is used to call sum_list function. The stack pointer %rsp is subtracted by 8 (1 byte) to save the address 0x1d of

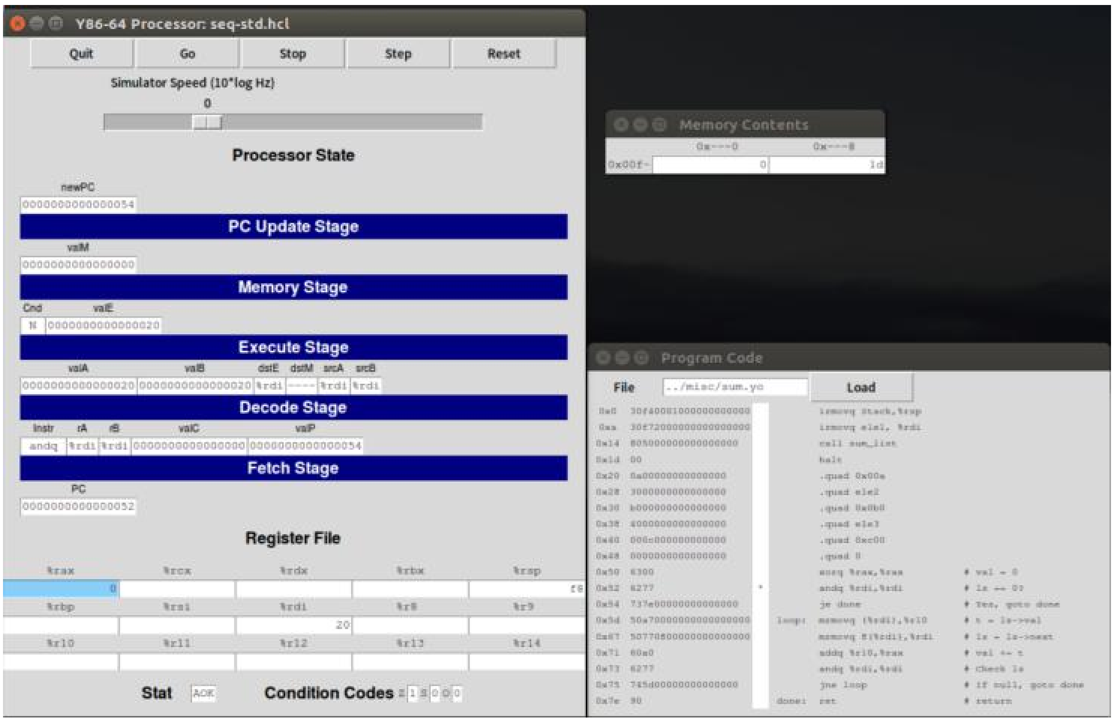
the next sequential execution instruction of the CALL instruction, and the PC is changed to the first address of sum_list function.



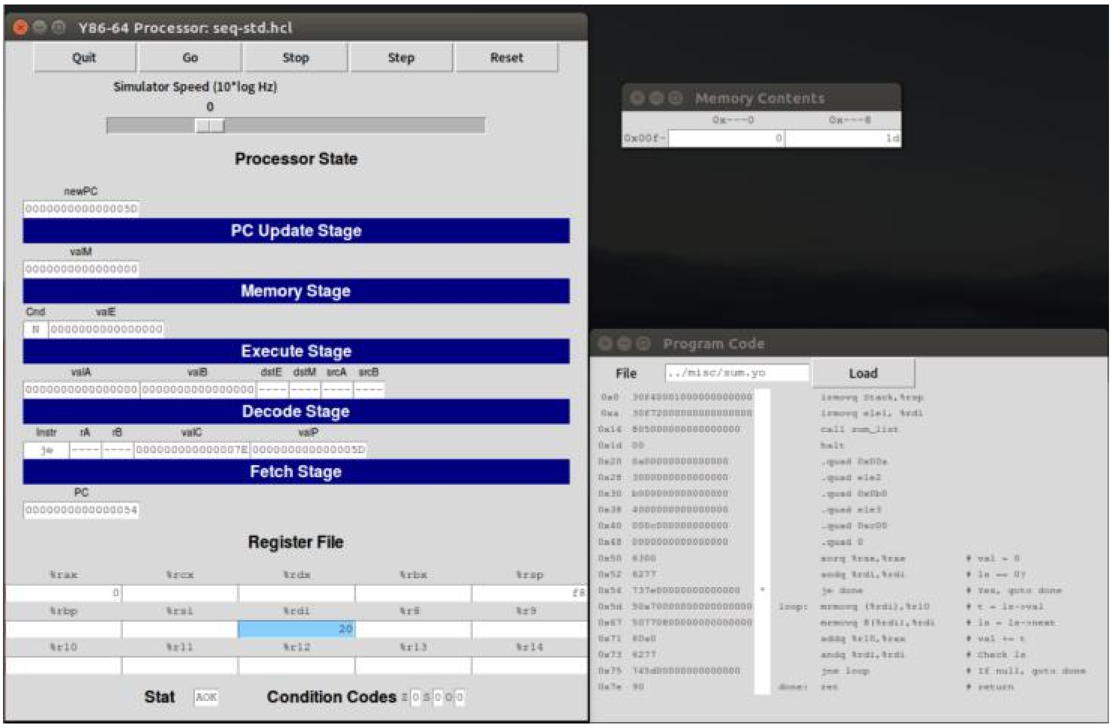
(4) This is an initialization step, the value in the %rax register (usually used as a return value register) is set to 0 using the XORQ instruction to hold the final sum_list result.



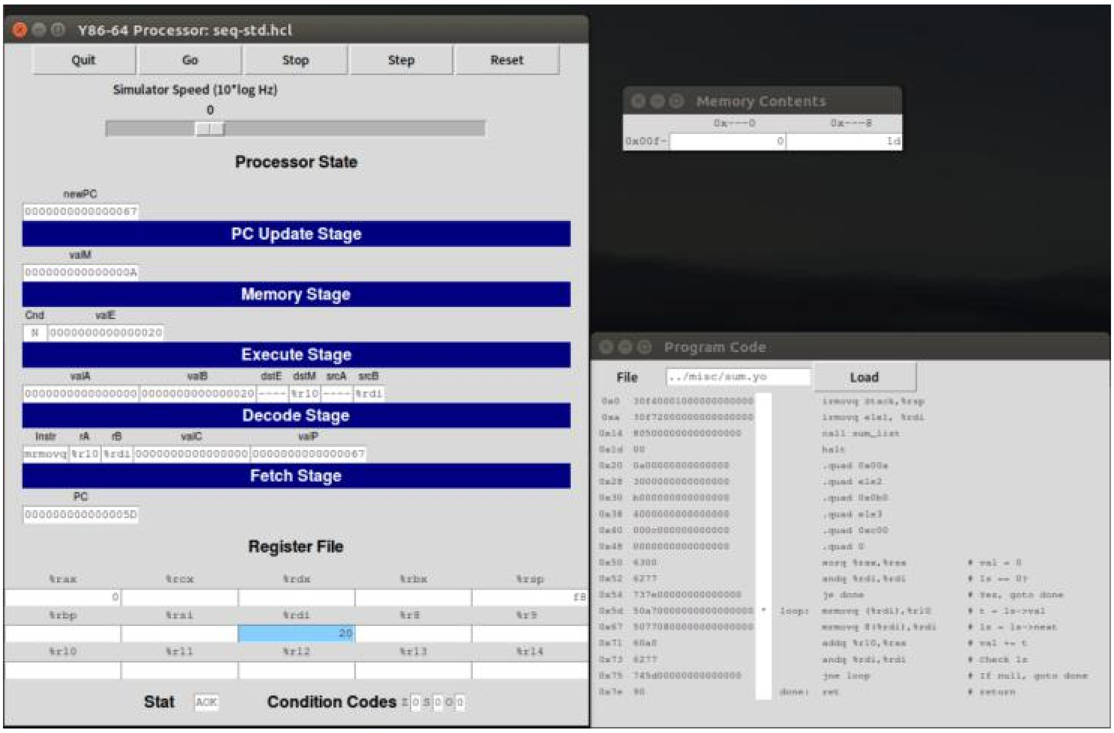
(5) The condition code register can be modified by the ANDQ instruction on the value of the register %rdi and itself. Since if %rdi is 0 and the result of the ANDQ instruction is 0 (NULL), the condition code ZF will be set to 1.



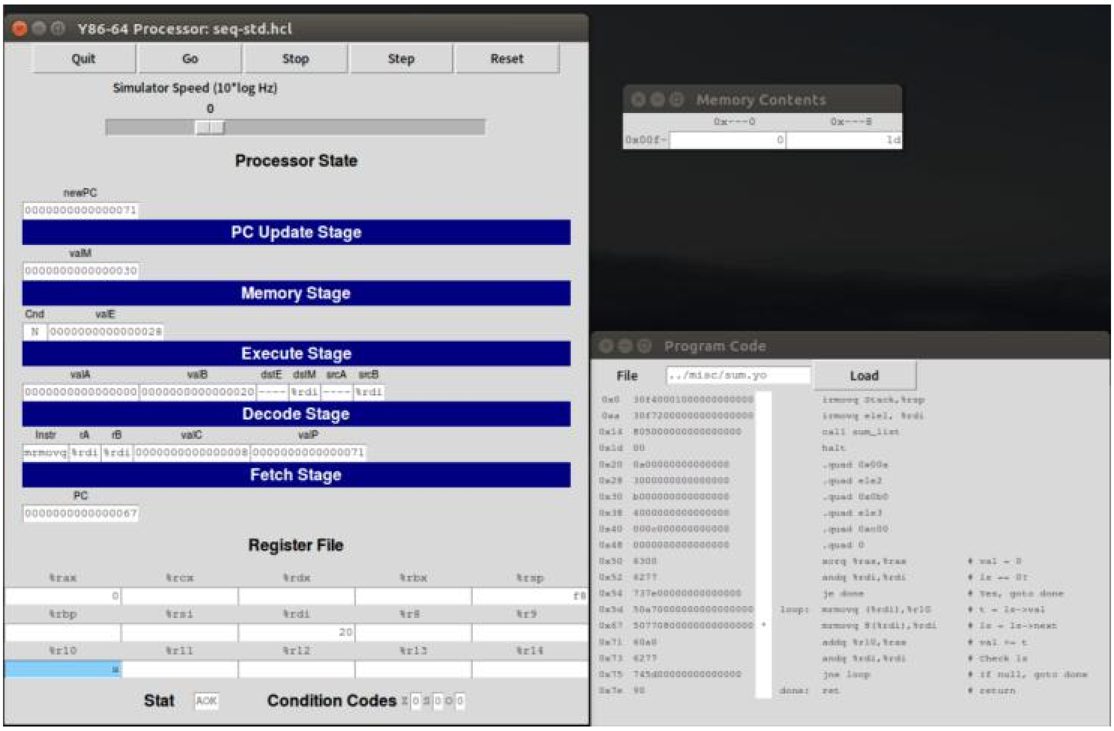
(6) The JE instruction is used to determine whether to jump to the address of label done pointed by JE by the value of condition code ZF. If ZF is 1 and Cnd condition is Y, the PC is set as the address of done tag; if Z is 0, the jump is not performed. Here the JE instruction will not execute.



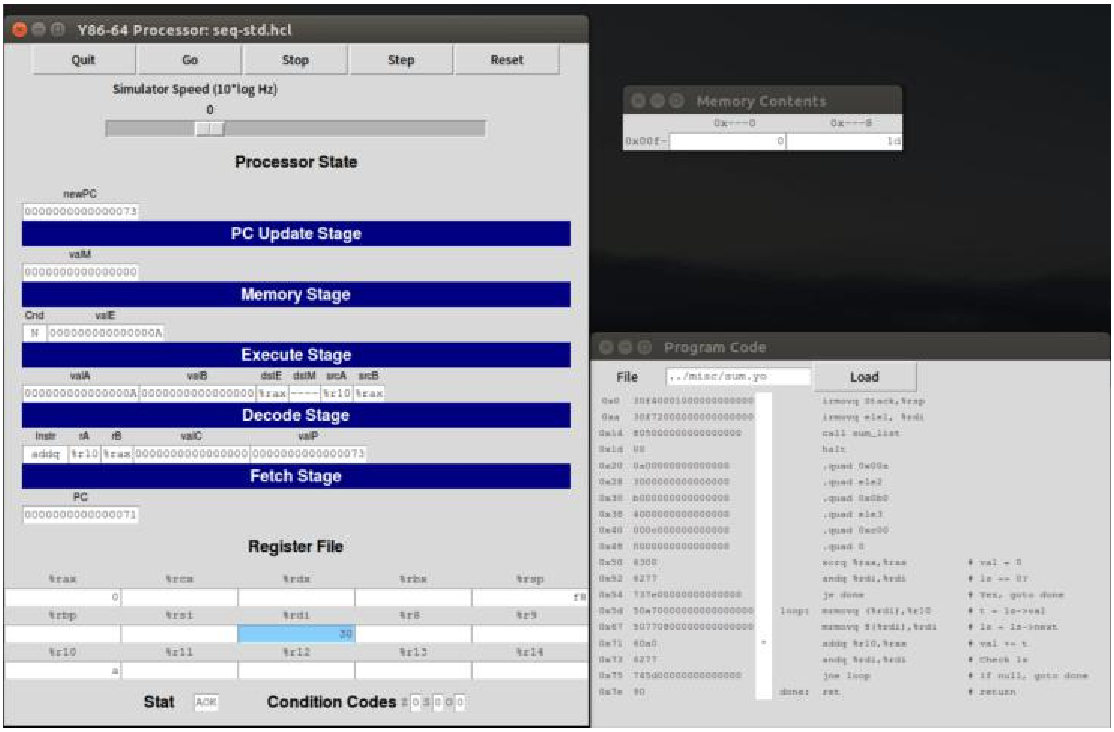
(7) Using the MRMOVQ instruction, use the value stored in %rdi as the address to find the corresponding value, and then put the found value into the %r10 register (take the value of the **first** node in the list as the addend).



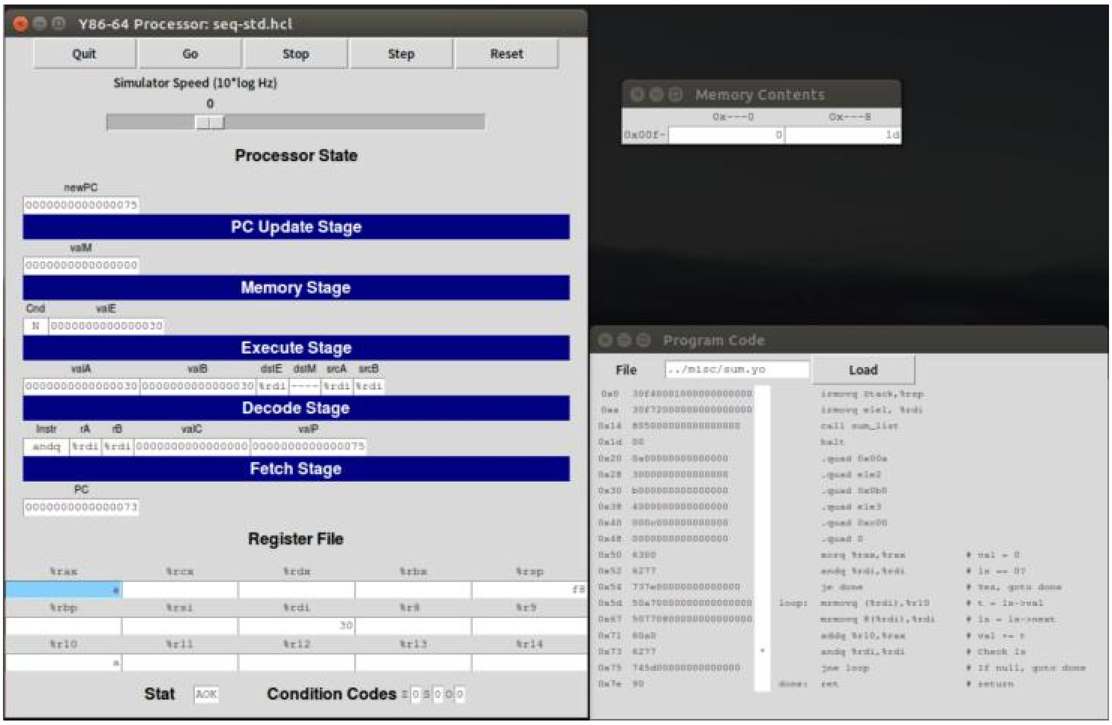
(8) Using the MRMOVQ instruction, the value + 8 (1 byte offset) stored in %rdi is used as the address to find the corresponding value, and then the found value is put into the %rdi register (take the address of next node of the current node).



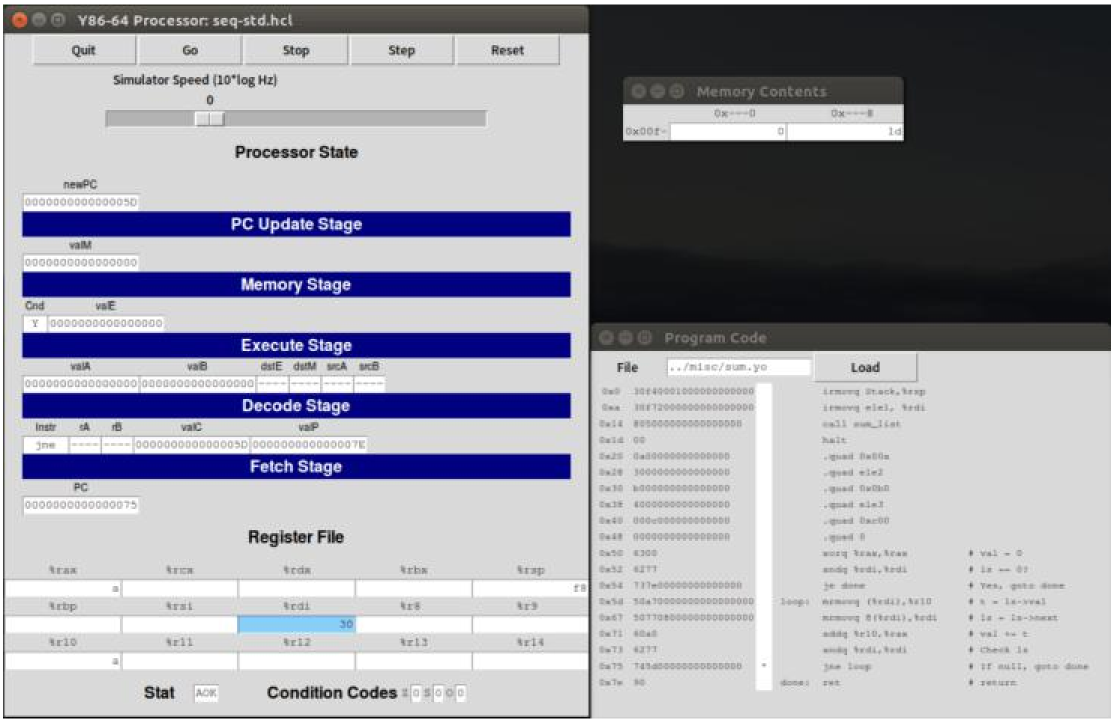
(9) Add register %rax to the value of register %r10 using the ADDQ instruction and store the result at %rax.



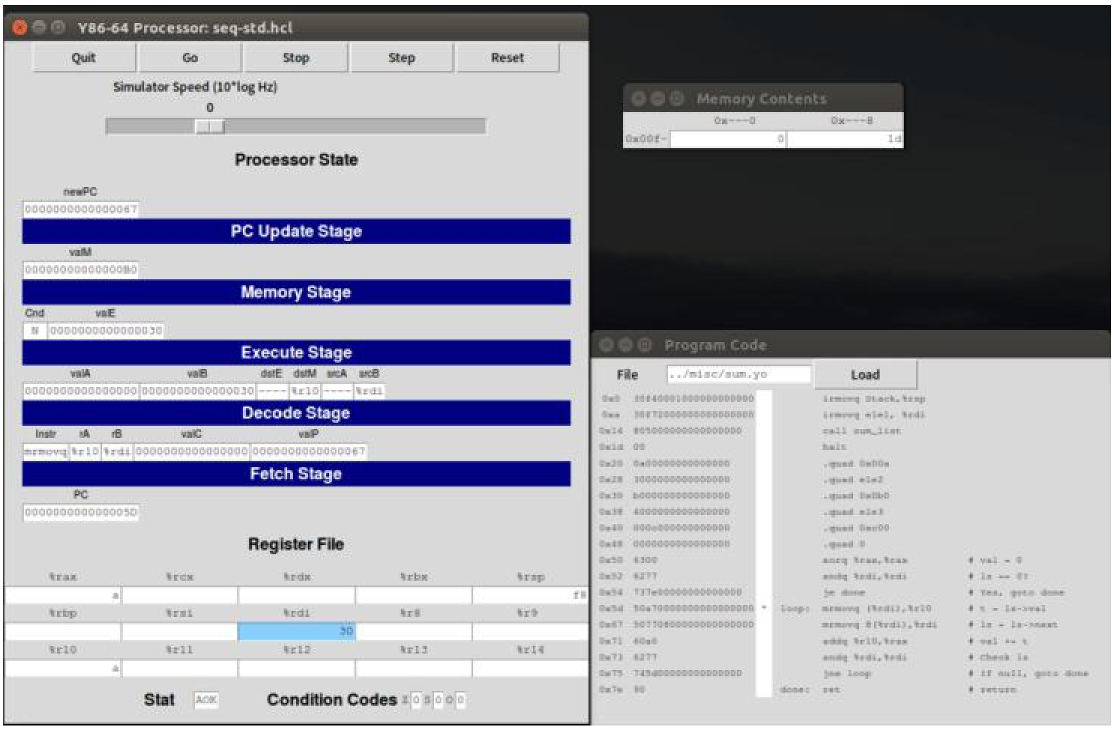
(10) The principle is the same as step (5). ANDQ instruction is used to determine whether the value (node address) stored in the current register %rdi is 0 (NULL), if so, the condition code ZF is set to 1.



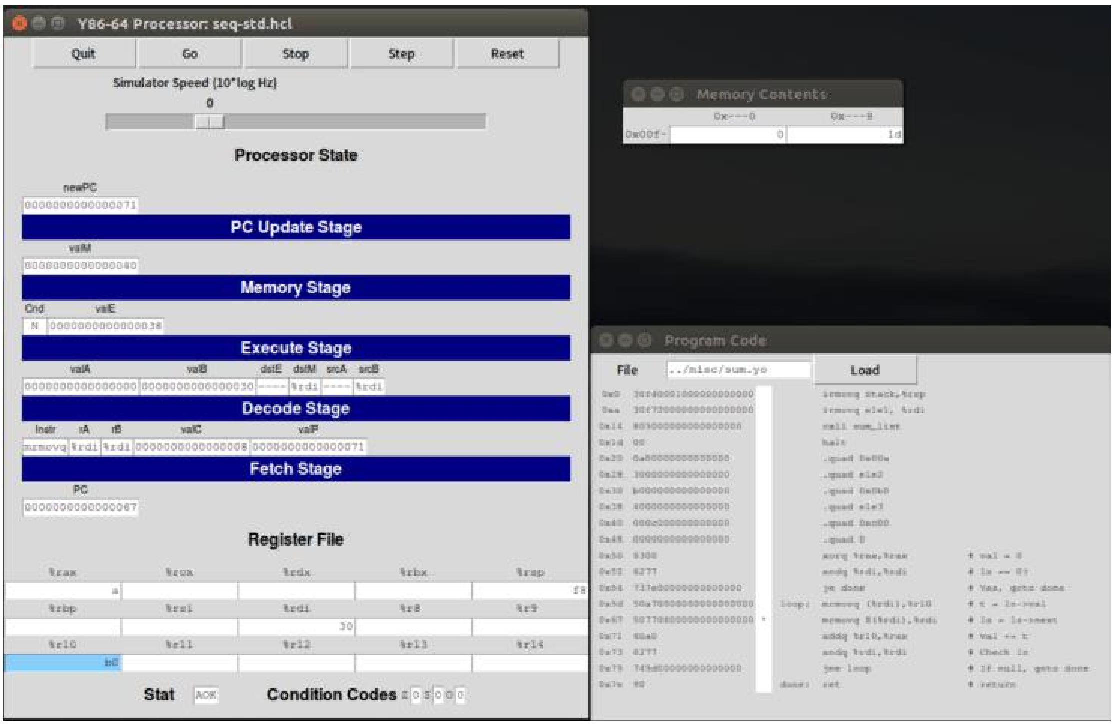
(11) The JNE instruction uses the condition code ZF result obtained by the ANDQ instruction in the previous step to decide whether to jump to the label loop position. In the current step, ZF is 0, and the condition for the execution of JNE instruction is that ZF is not equal to 1, the Cnd condition is Y, so the current PC is updated to the instruction address of the label loop.



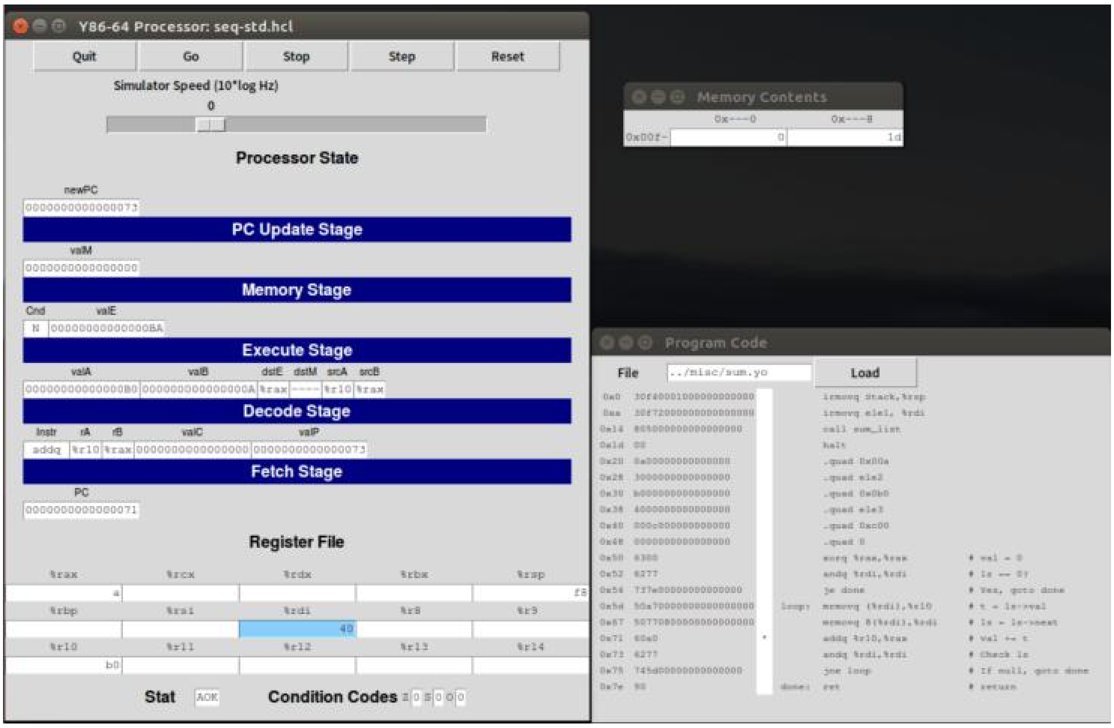
(12) Go back to the loop and execute MRMOVQ instruction again, use the value stored in %rdi as the address to find the corresponding value, and then put the found value into the %r10 register (take the value of the **second** node in the list as the addend).



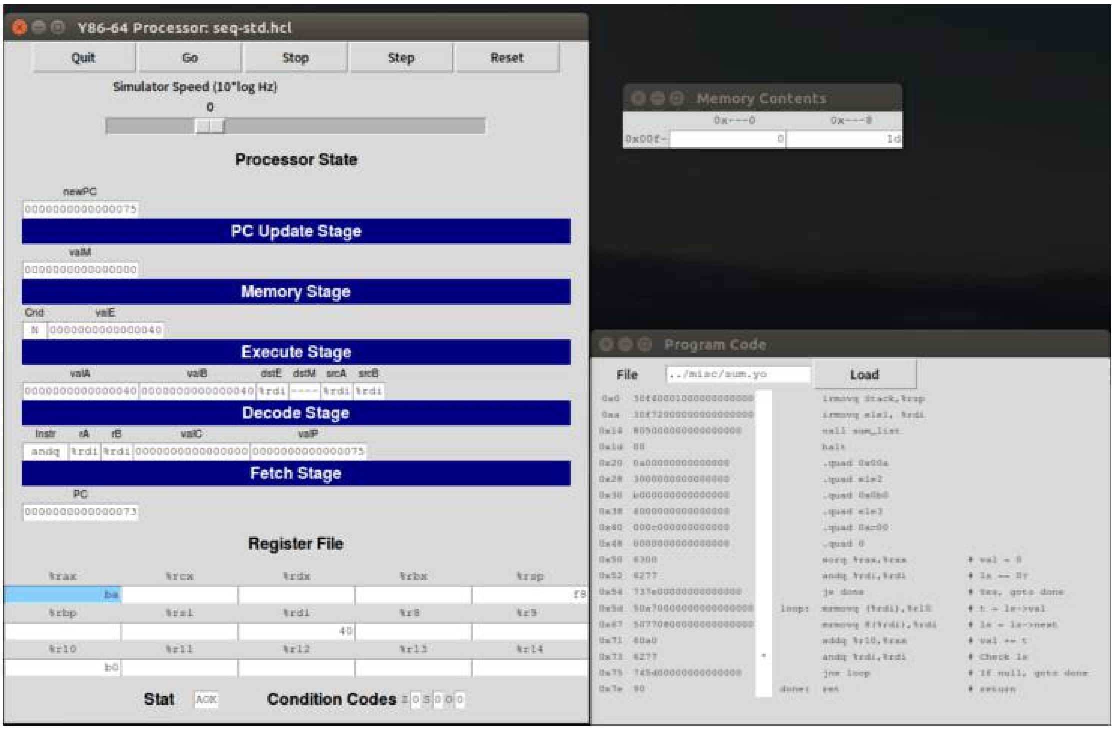
(13) Using the MRMOVQ instruction again, the value + 8 (1 byte offset) stored in %rdi is used as the address to find the corresponding value, and then the found value is put into the %rdi register (take the address of next node of the current node).



(14) Add register %rax to the value of register %r10 using the ADDQ instruction and store the result at %rax.

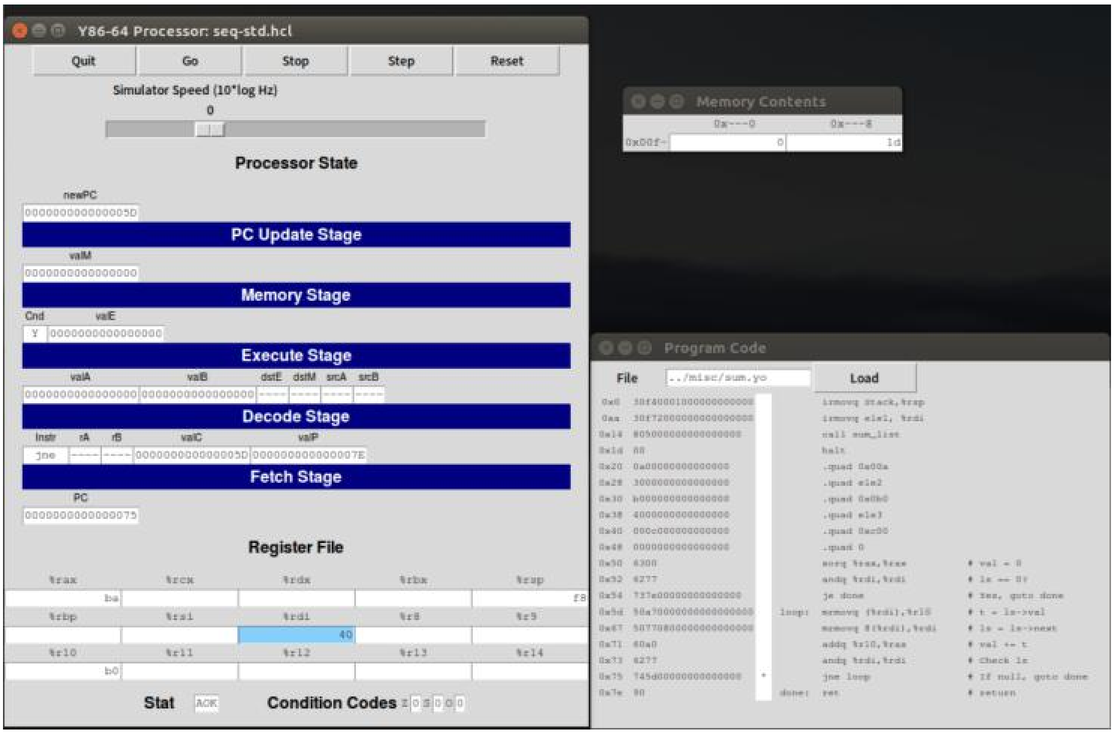


(15) Use ANDQ instruction again to determine whether the value (node address) stored in the current register %rdi is 0 (NULL), if so, the condition code ZF is set to 1.

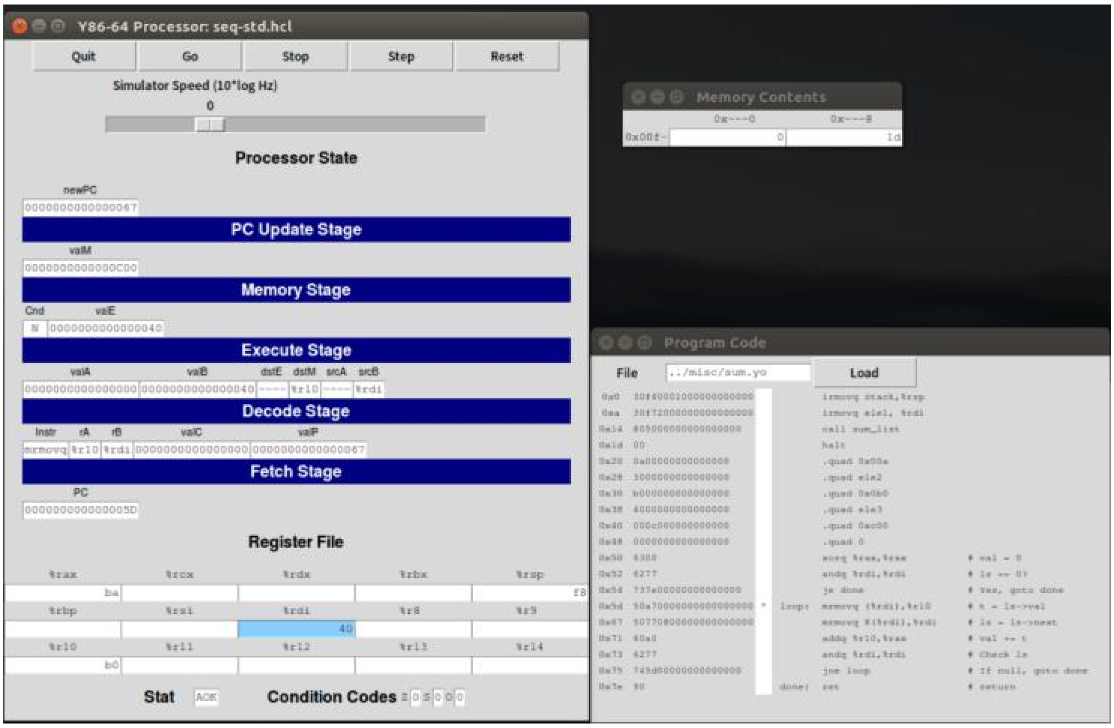


(16) In the current step, ZF is 0, and the condition for the execution of

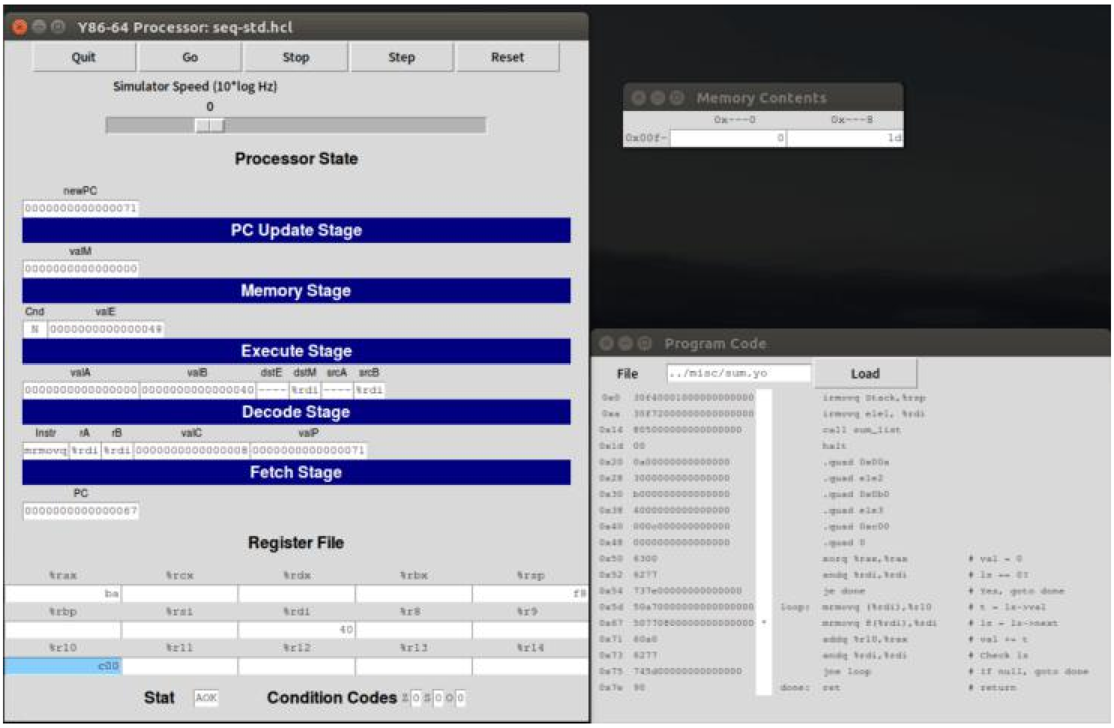
JNE instruction is that ZF is not equal to 1, the Cnd condition is Y, so the current PC is updated to the instruction address of the label loop.



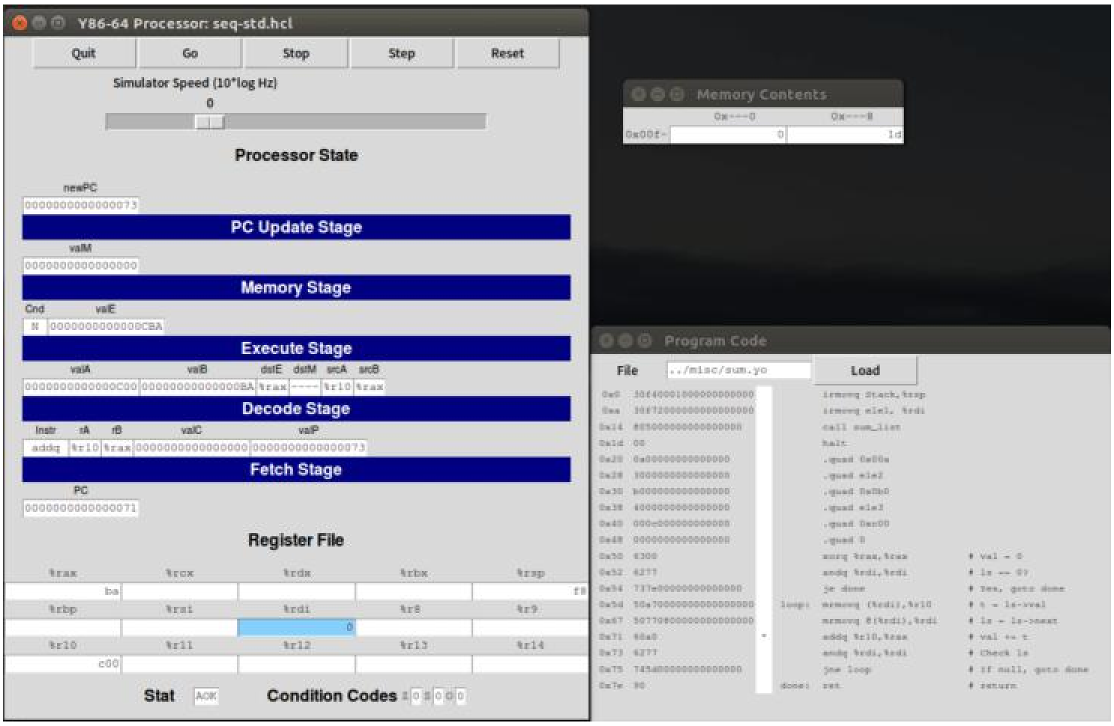
(17) Go back to the loop and execute MRMOVQ instruction again, use the value stored in %rdi as the address to find the corresponding value, and then put the found value into the %r10 register (take the value of the **third** node in the list as the addend).



(18) Using the MRMOVQ instruction again, the value + 8 (1 byte offset) stored in %rdi is used as the address to find the corresponding value, and then the found value is put into the %rdi register (take the address of next node of the current node). And we can see that the next address is NULL(0).

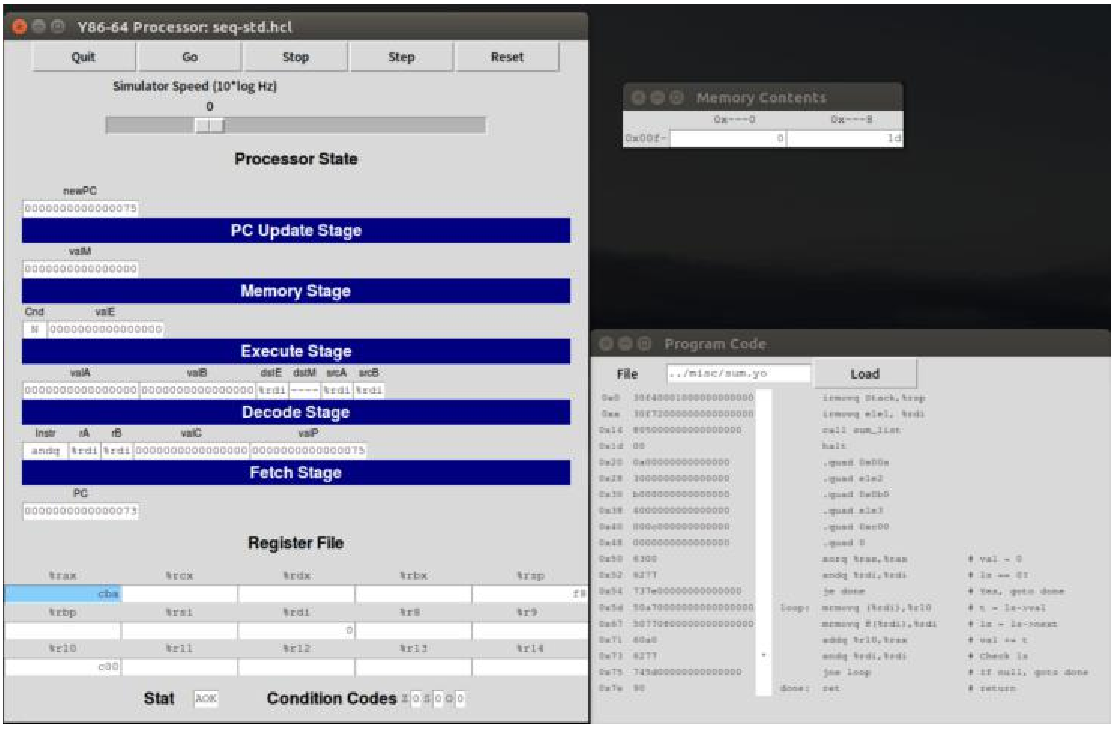


(19) Add register %rax to the value of register %r10 using the ADDQ instruction and store the result at %rax.

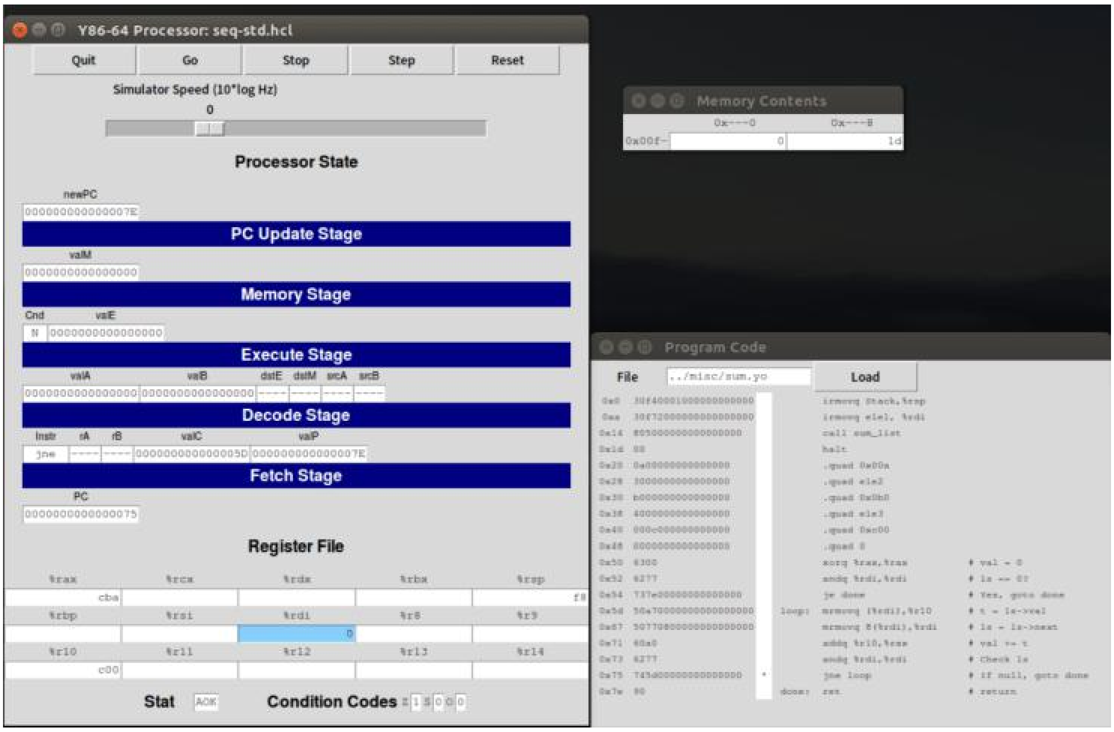


(20) Use ANDQ instruction again to determine whether the value (node address) stored in the current register %rdi is 0 (NULL), if so,

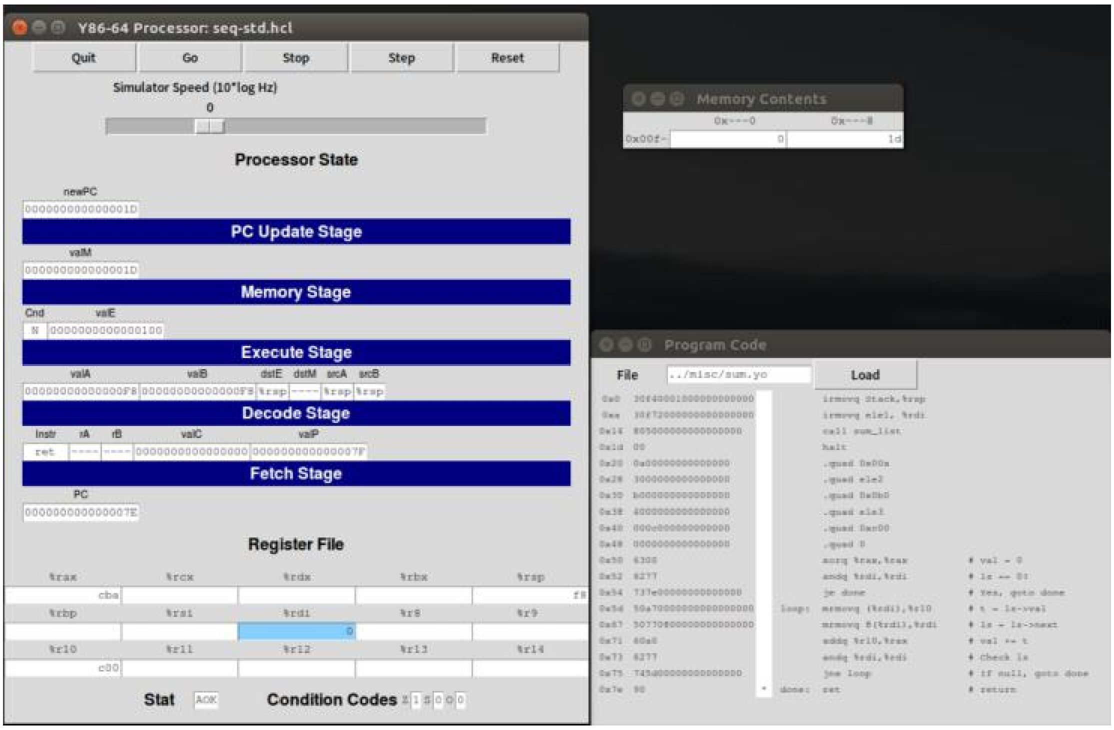
the condition code ZF is set to 1, here the ZF will be set to 1.



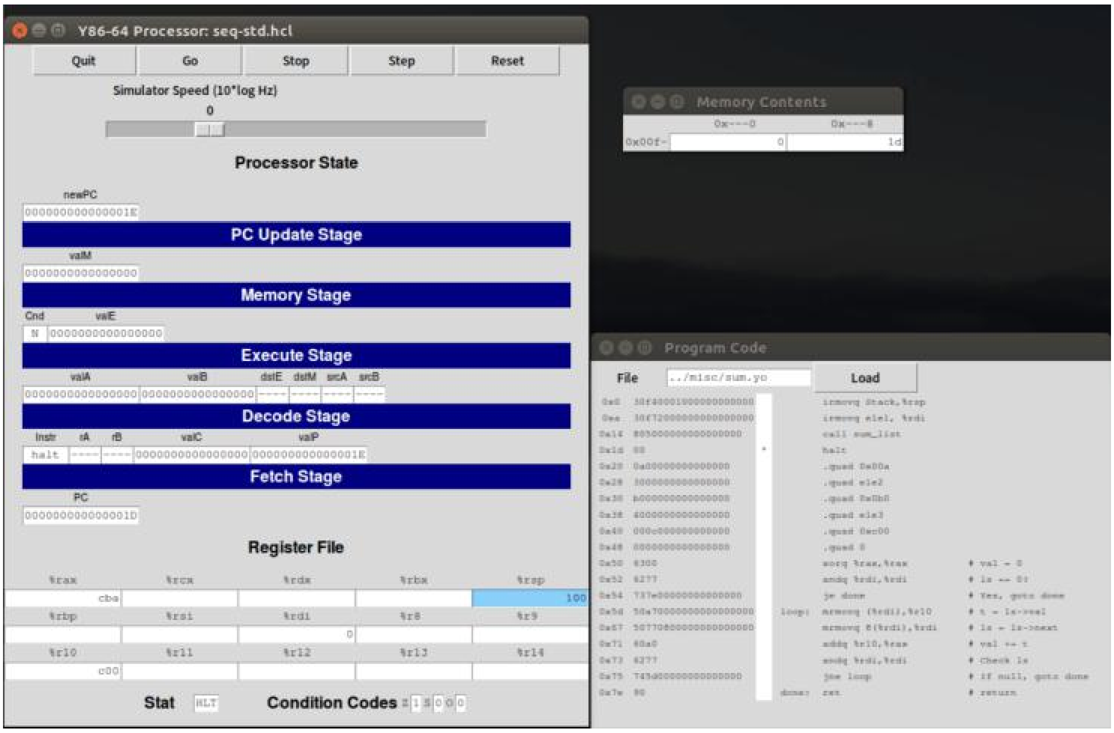
(21) In the current step, ZF is 1 and the condition for the execution of JNE instruction is that ZF is not equal to 1, the Cnd condition is N, this JNE instruction will not execute, so the current PC is updated with the address of the next sequential execution instruction.



(22) Execute the RET instruction, representing the end of function execution, return the execution result saved in register %rax(0xcba), PC will be updated with the instruction address (0x1d) of the stack when the CALL instruction was issued last time, and the stack pointer + 8(1 byte) indicates that the stack element is poped.

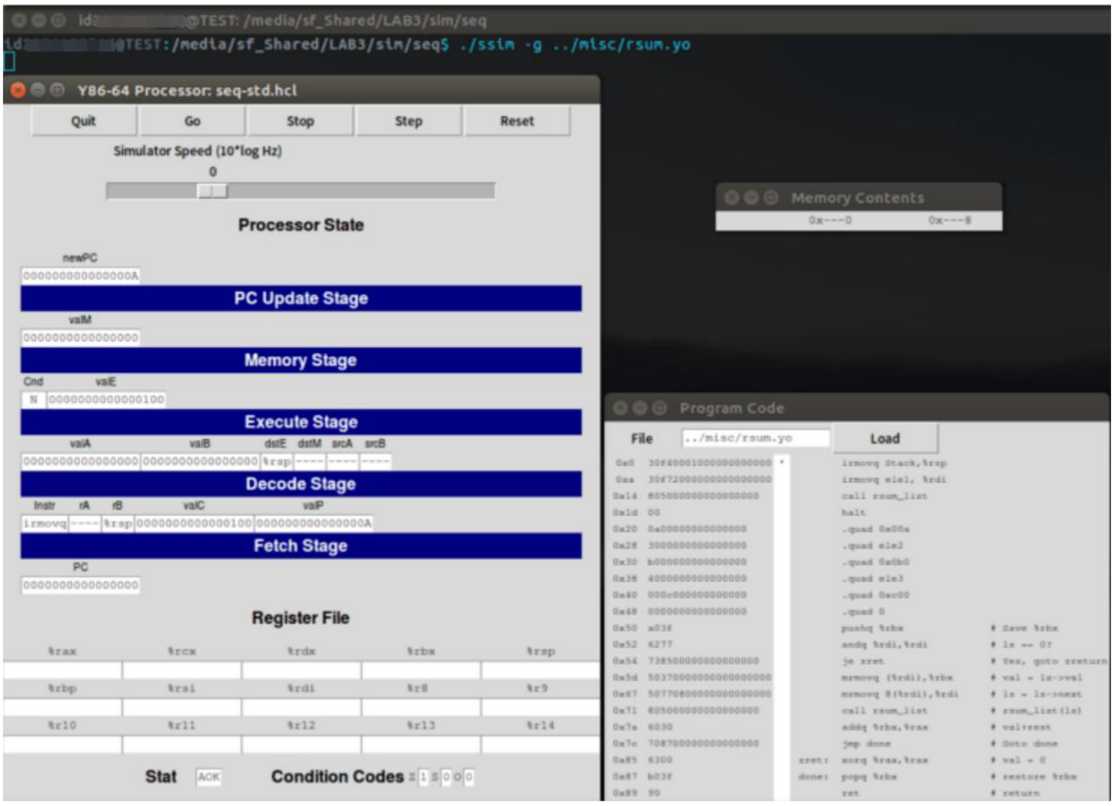


(23) Jump to 0x1d, execute HALT instruction, change the execution status of the program from AOK to HLT, indicating that the program stops running.

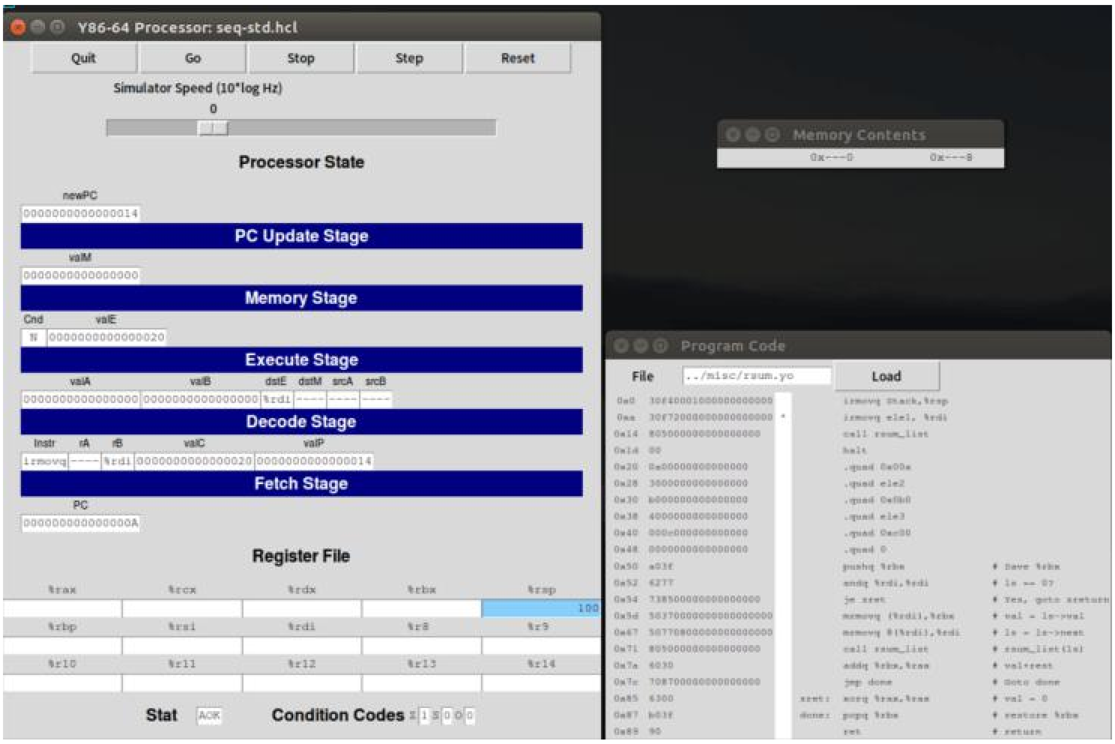


- Results and Analysis for **rsum.yo**

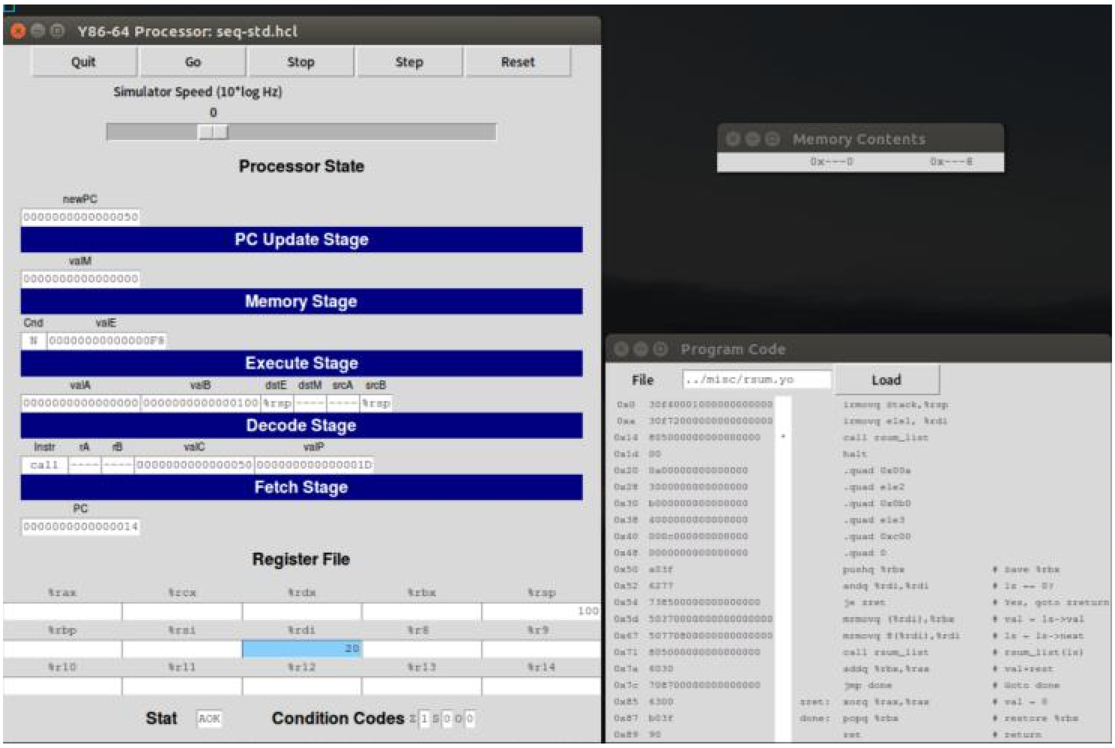
(1) Initialize the stack pointer so that it points to 0x100.



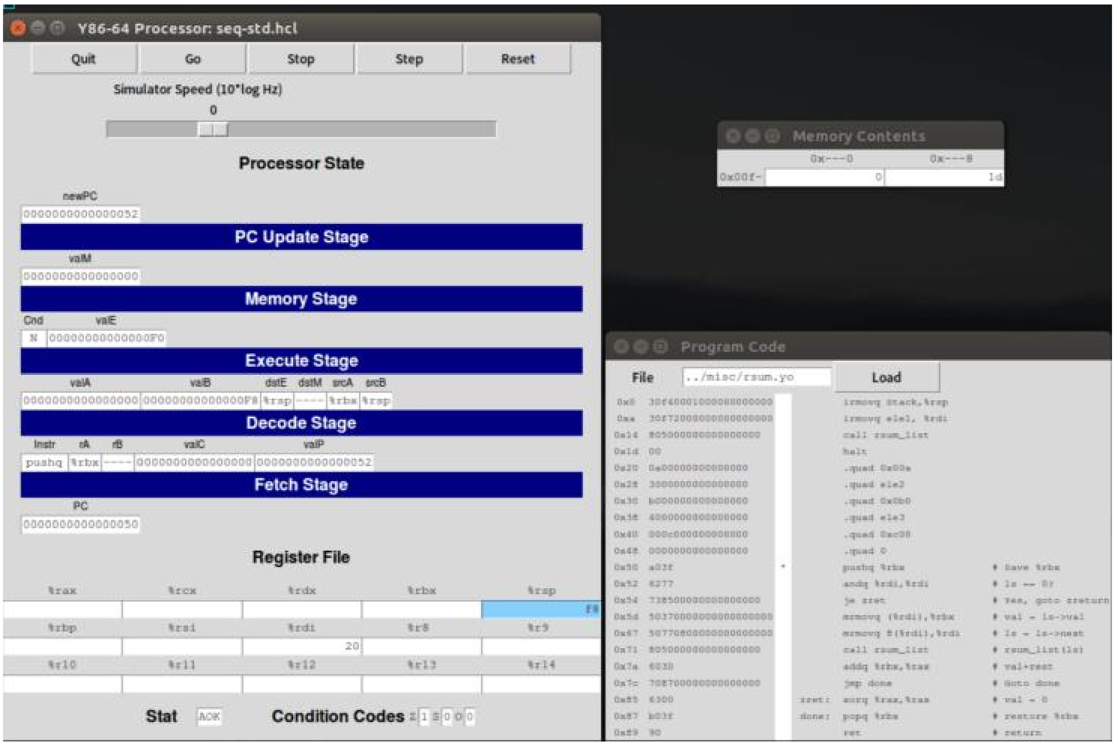
(2) Use IRMOVQ instruction to put the first node address of the linked list into the %rdi register.



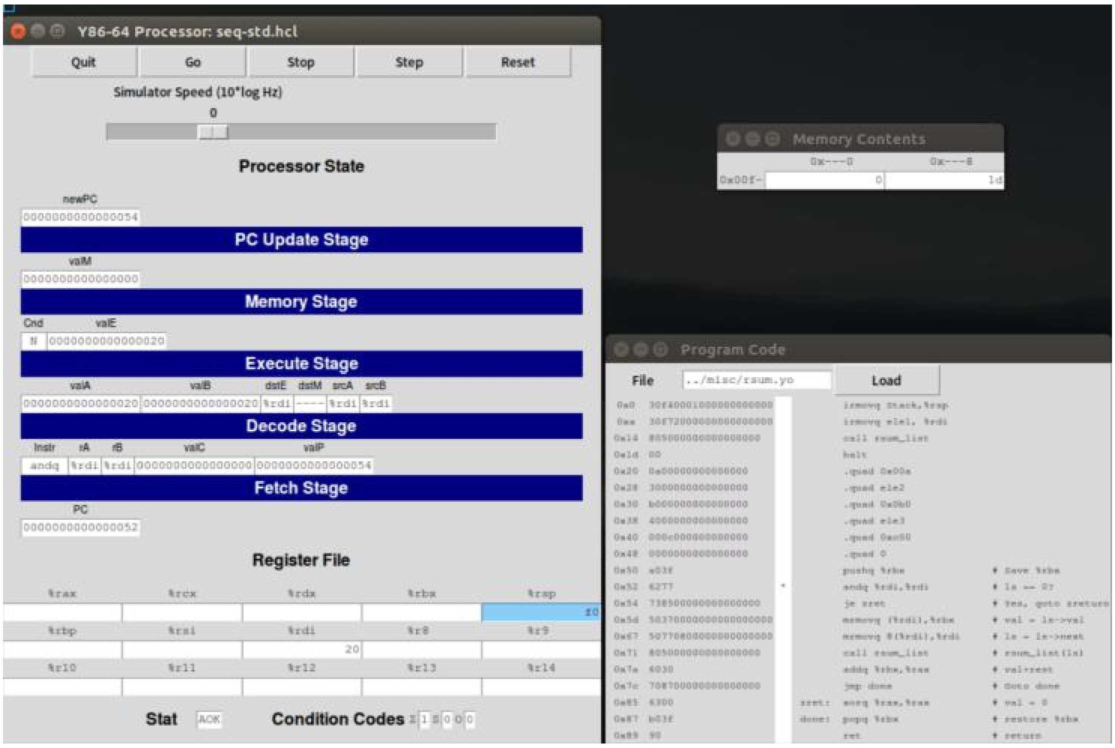
(3) The CALL instruction is used to call rsum_list function. The stack pointer %rsp is subtracted by 8 (1 byte) to save the address 0x1d of the next sequential execution instruction of the CALL instruction, and the PC is changed to the first address of rsum_list function(0x50).



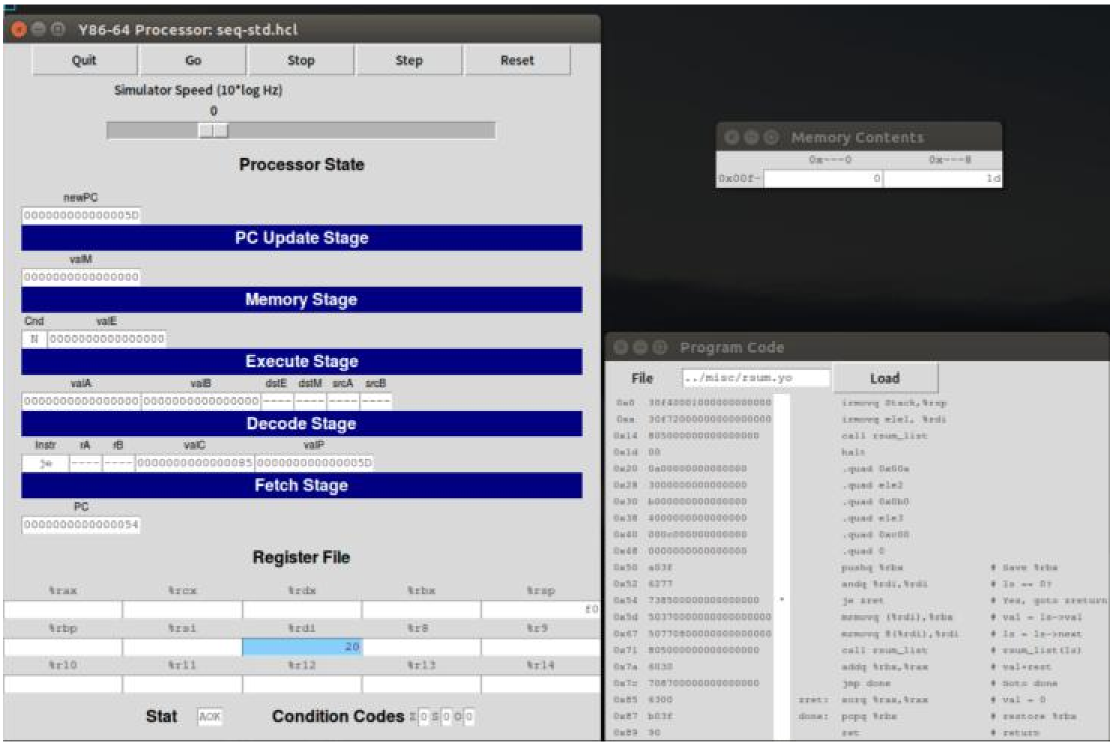
(4) Use the PUSHQ instruction to push the value stored in register %rbx. In this program, we use register %rbx to store value from the link list recursively. But in the current step, %rbx has not been assigned, and only the value of register %rsp is subtracted by 8 (1 byte).



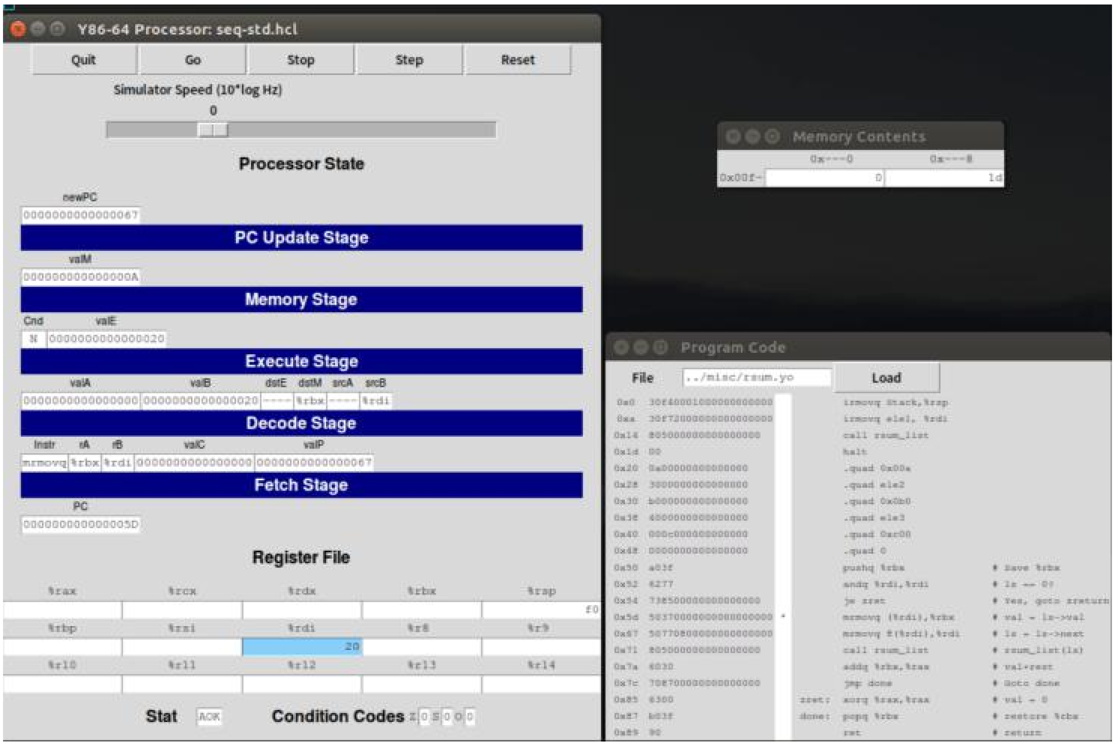
(5) The condition code register can be modified by the ANDQ instruction on the value of the register %rdi and itself. Since if %rdi is 0 and the result of the ANDQ instruction is 0 (NULL), the condition code ZF will be set to 1.



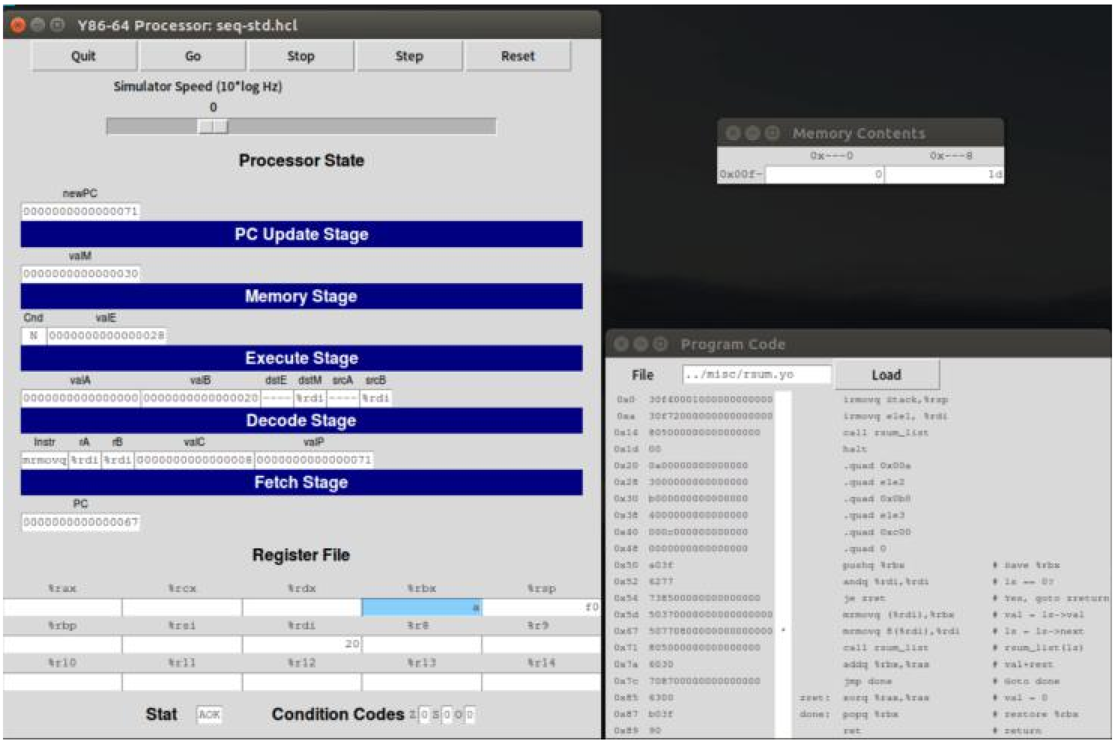
(6) The JE instruction is used to determine whether to jump to the address of label zret pointed by JE by the value of condition code ZF. If ZF is 1 and Cnd condition is Y, the PC is set as the address of done tag; if Z is 0, the jump is not performed. Here the JE instruction will not execute.



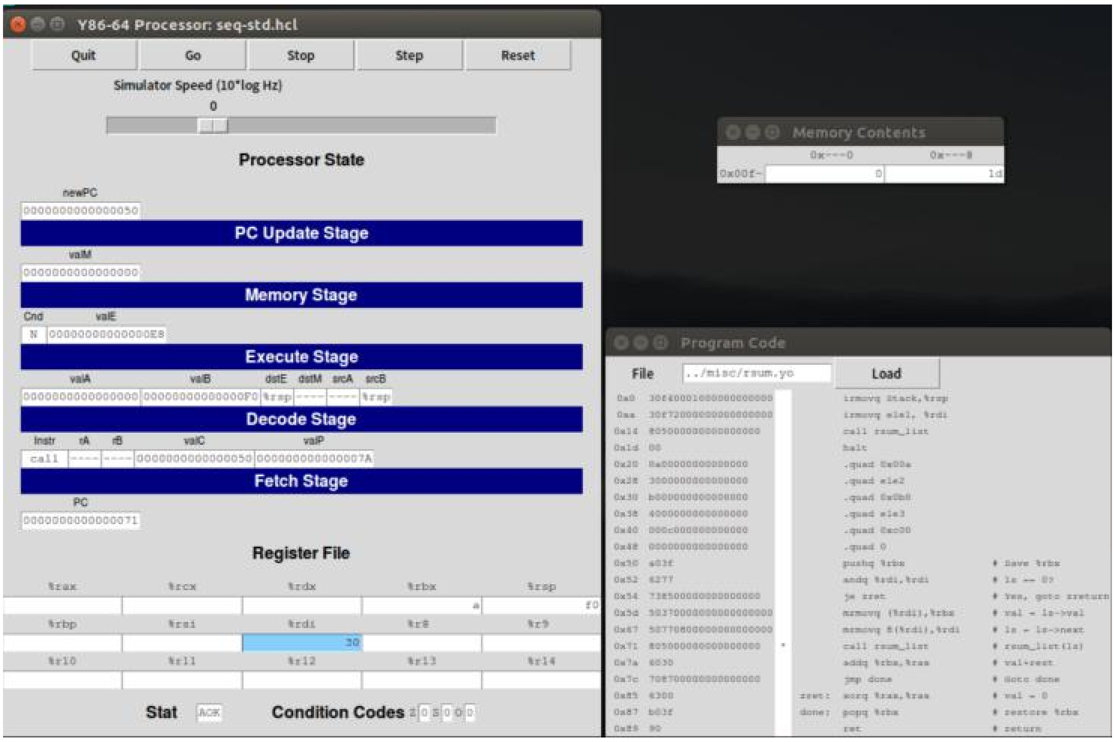
(7) Using the MRMOVQ instruction, use the value stored in %rdi as the address to find the corresponding value, and then put the found value into the %rbx register (take the value of the **first** node in the list as the addend).



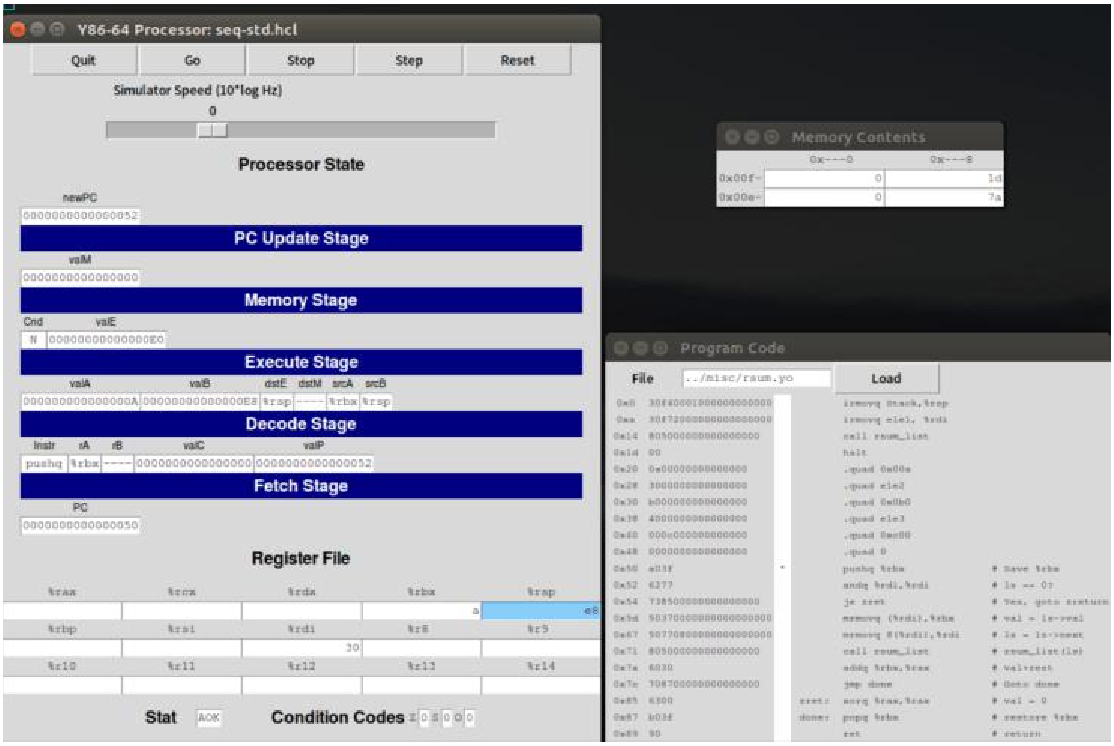
(8) Using the MRMOVQ instruction, the value + 8 (1 byte offset) stored in %rdi is used as the address to find the corresponding value, and then the found value is put into the %rdi register (take the address of next node of the current node).



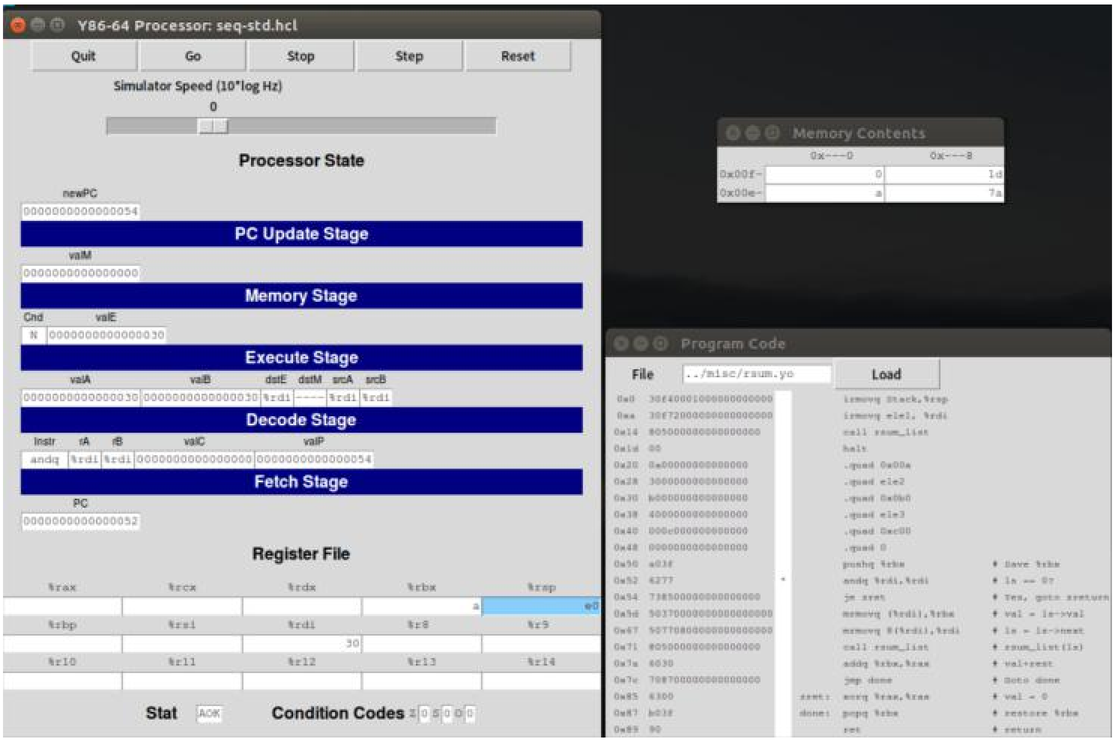
(9) (First recursion) The rsum_list function is called recursively. First, subtract the stack pointer %rsp by 8(1 byte), and then the address 0x7a of the original next execution address is stored to the location pointed by %rsp. PC is updated to the starting address of rsum_list function.



(10) Use the PUSHQ instruction to push the value stored in register %rbx. And in the current step, %rbx has stored the value 0xa of the first node in the linked list stored when rsum_list was last called, hence, the stack pointer %rsp will be subtracted by 8 and store 0xa in where it points to.

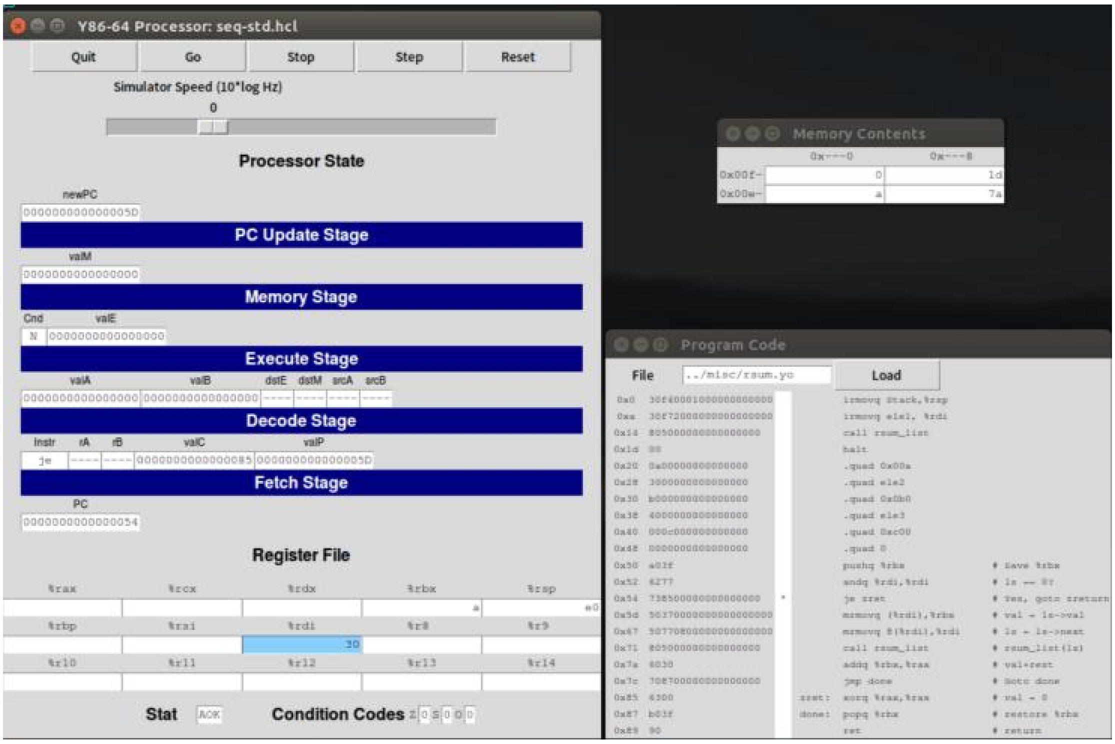


(11) Again, use the ANDQ instruction to check whether the value of %rdi is NULL (0), if %rdi is 0 and the result of the ANDQ instruction is 0 (NULL), the condition code ZF will be set to 1.

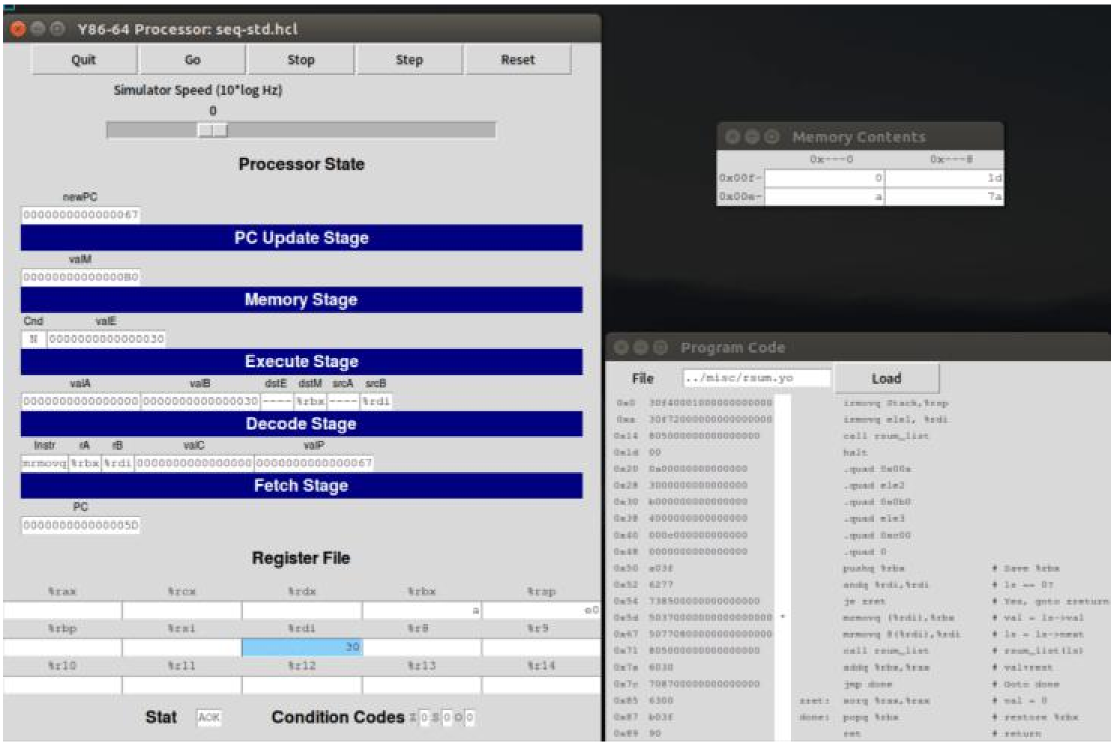


(12) The JE instruction is used again to determine whether to jump

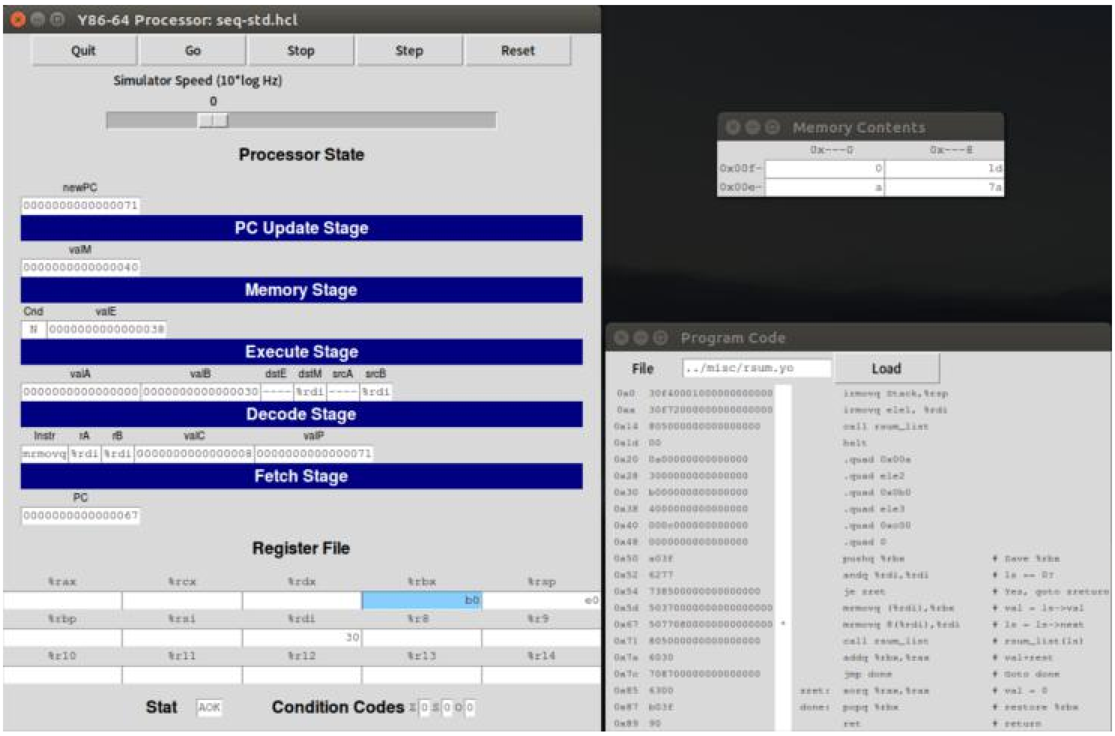
according to the condition code ZF set in the previous ANDQ instruction. The current register %rdi is non-0, ZF is 0, and Cnd is N. Therefore, the contents of JE instruction are not executed, and the PC is only updated to the address of the next instruction executed in sequence.



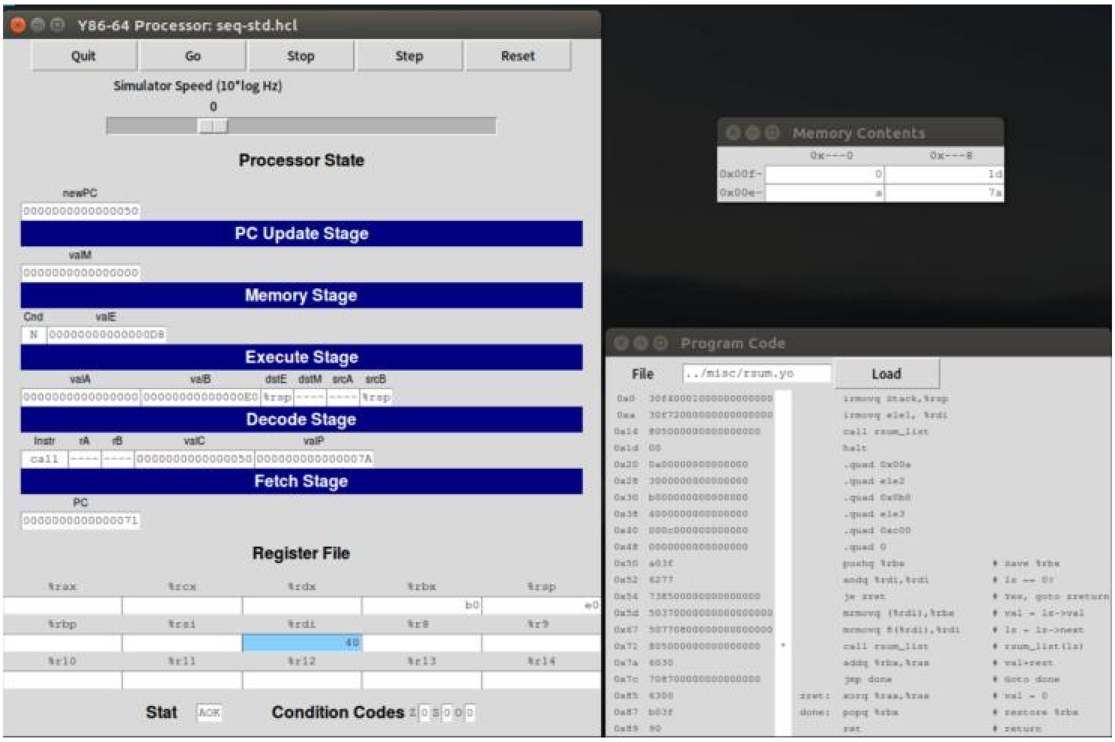
(13) Using the MRMOVQ instruction, use the value stored in %rdi as the address to find the corresponding value, and then put the found value into the %rbx register (take the value of the **second** node in the list as the addend).



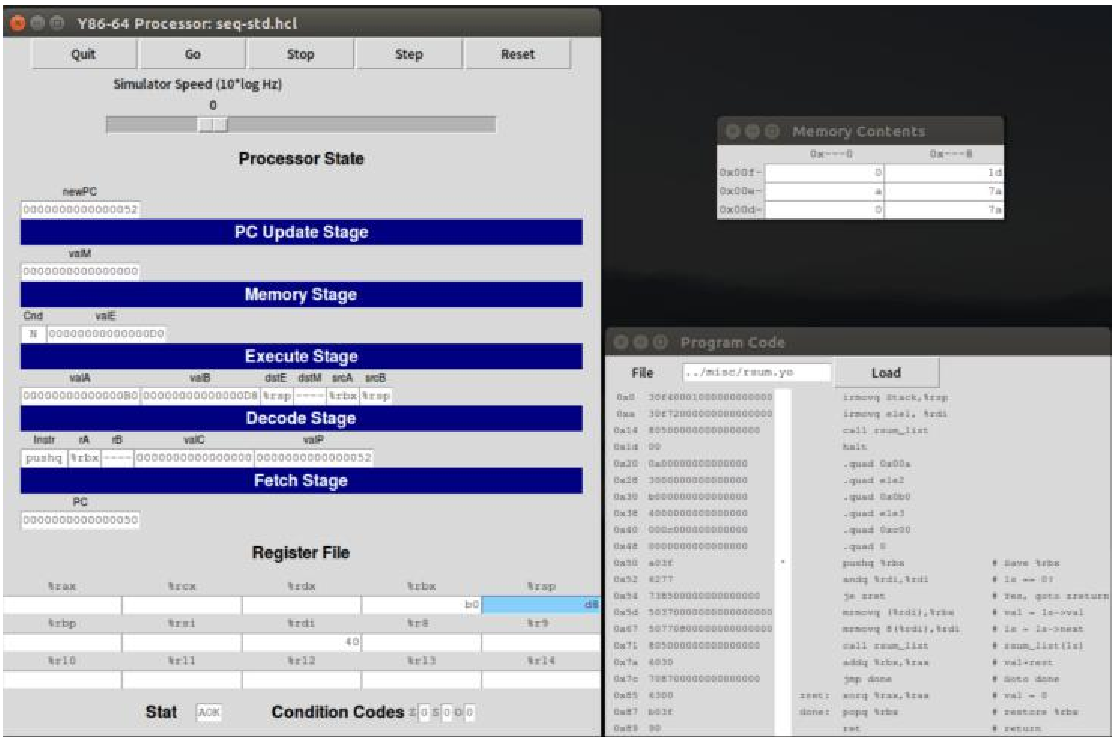
(14) Using the MRMOVQ instruction, the value + 8 (1 byte offset) stored in %rdi is used as the address to find the corresponding value, and then the found value is put into the %rdi register (take the address of next node of the current node).



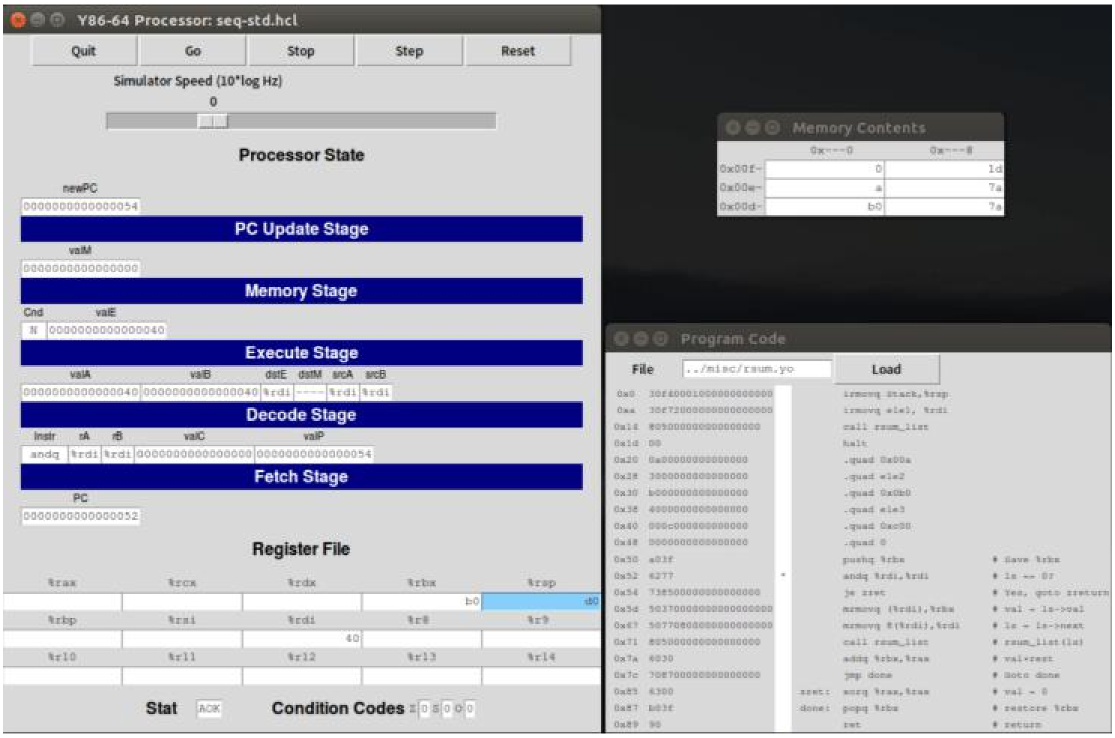
(15) (Second recursion) The rsum_list function is called recursively. First, subtract the stack pointer %rsp by 8(1 byte), and then the address 0x7a of the original next execution address is stored to the location pointed by %rsp. PC is updated to the starting address of rsum_list function.



(16) Use the PUSHQ instruction to push the value stored in register %rbx. And in the current step, %rbx has stored the value 0xb0 of the first node in the linked list stored when rsum_list was last called, hence, the stack pointer %rsp will be subtracted by 8 and store 0xb0 in where it points to.

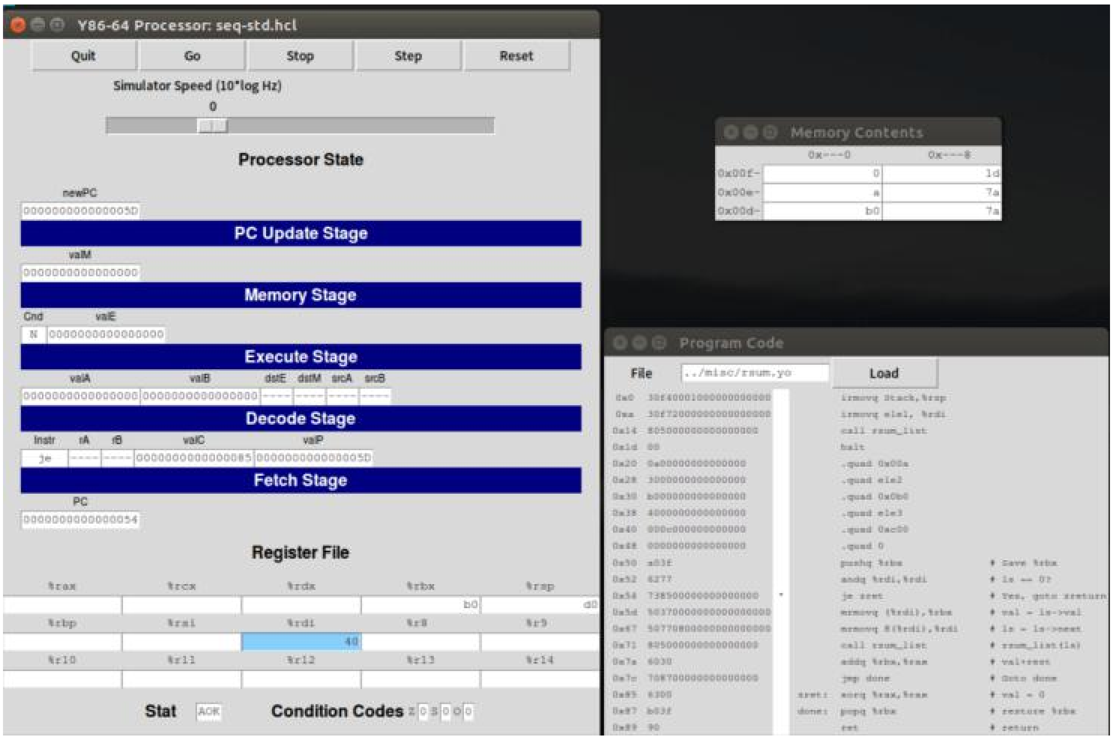


(17) Again, use the ANDQ instruction to check whether the value of %rdi is NULL (0), if %rdi is 0 and the result of the ANDQ instruction is 0 (NULL), the condition code ZF will be set to 1.

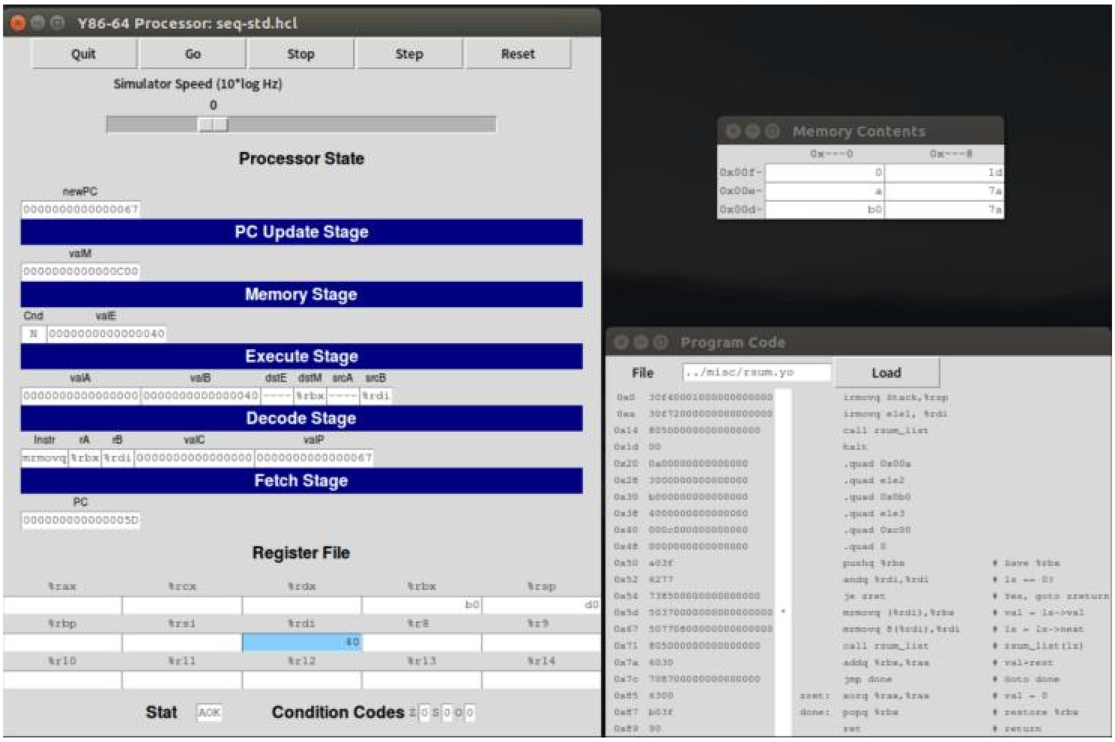


(18) The JE instruction is used again to determine whether to jump

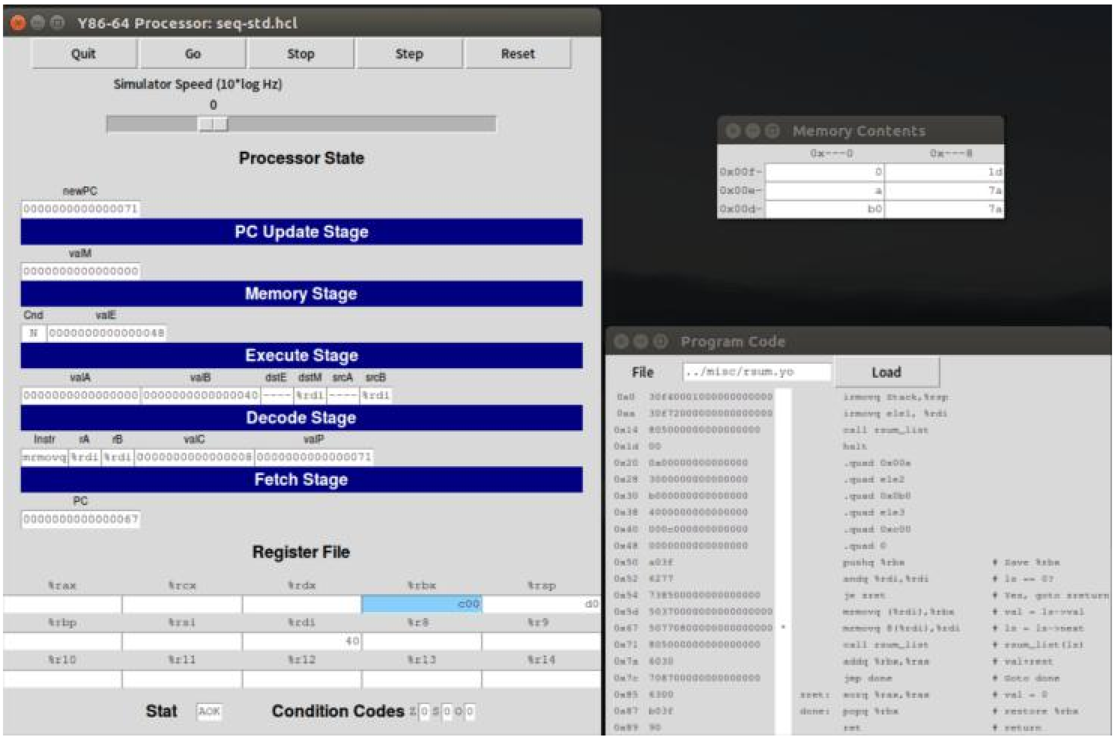
according to the condition code ZF set in the previous ANDQ instruction. The current register %rdi is non-0, ZF is 0, and Cnd is N. Therefore, the contents of JE instruction are not executed, and the PC is only updated to the address of the next instruction executed in sequence.



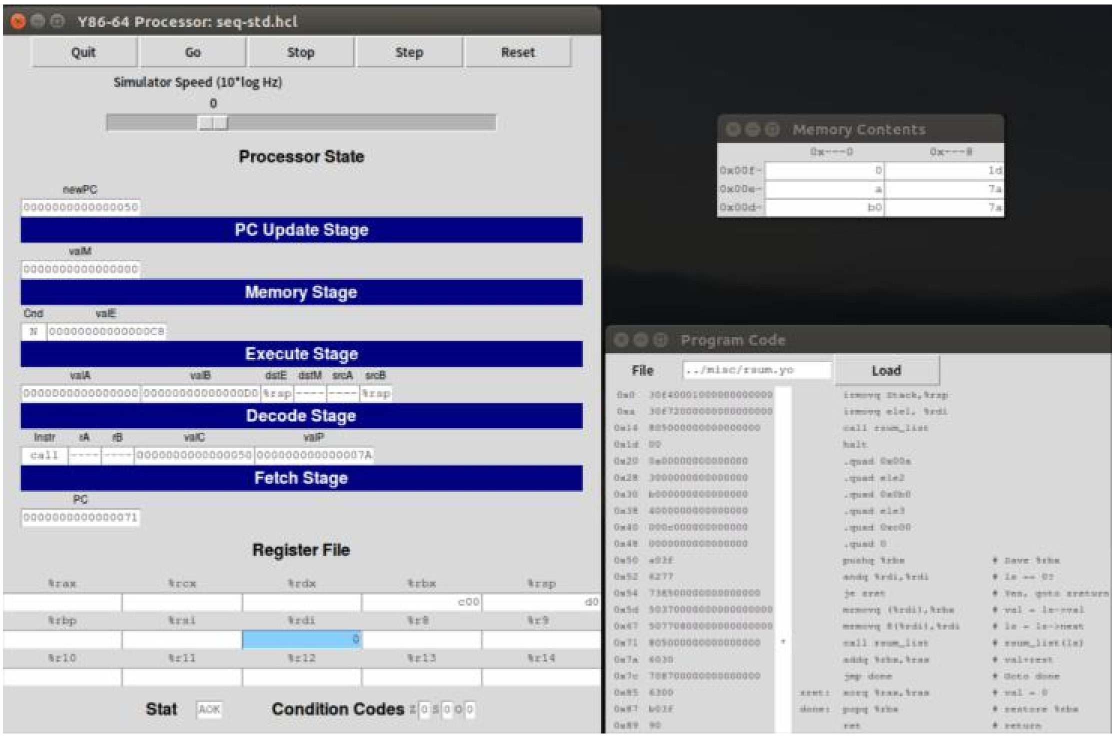
(19) Using the MRMOVQ instruction, use the value stored in %rdi as the address to find the corresponding value, and then put the found value into the %rbx register (take the value of the **third** node in the list as the addend).



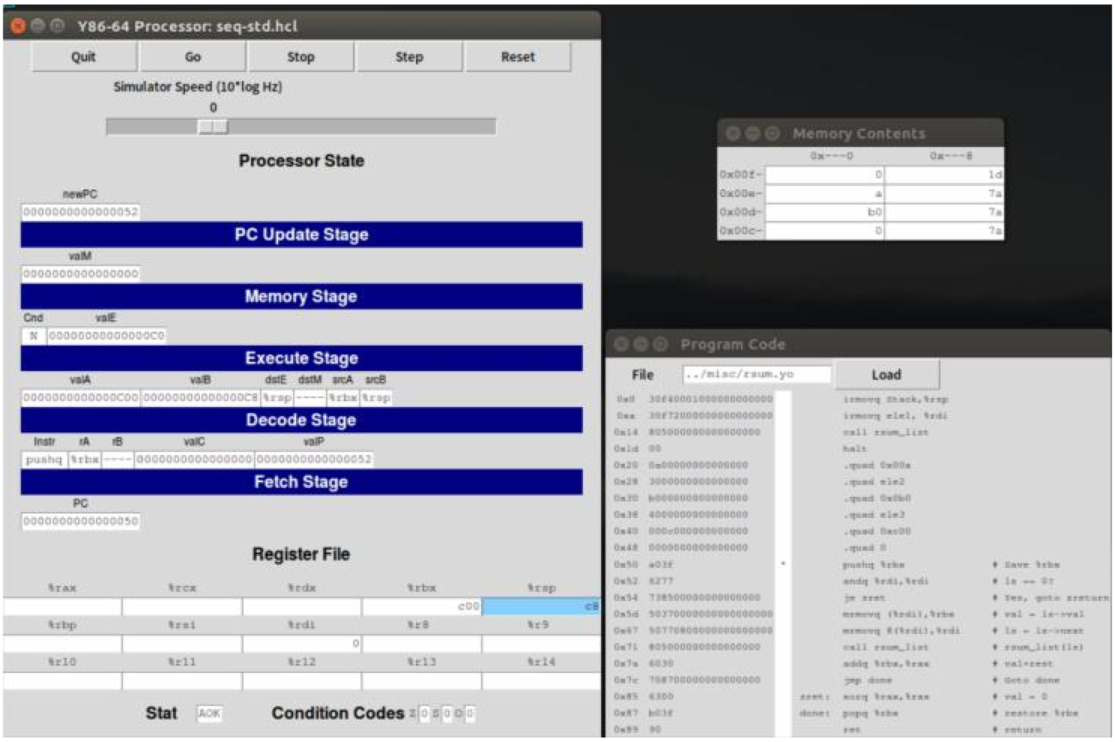
(20) Using the MRMOVQ instruction, the value + 8 (1 byte offset) stored in %rdi is used as the address to find the corresponding value, and then the found value is put into the %rdi register (take the address of next node of the current node).



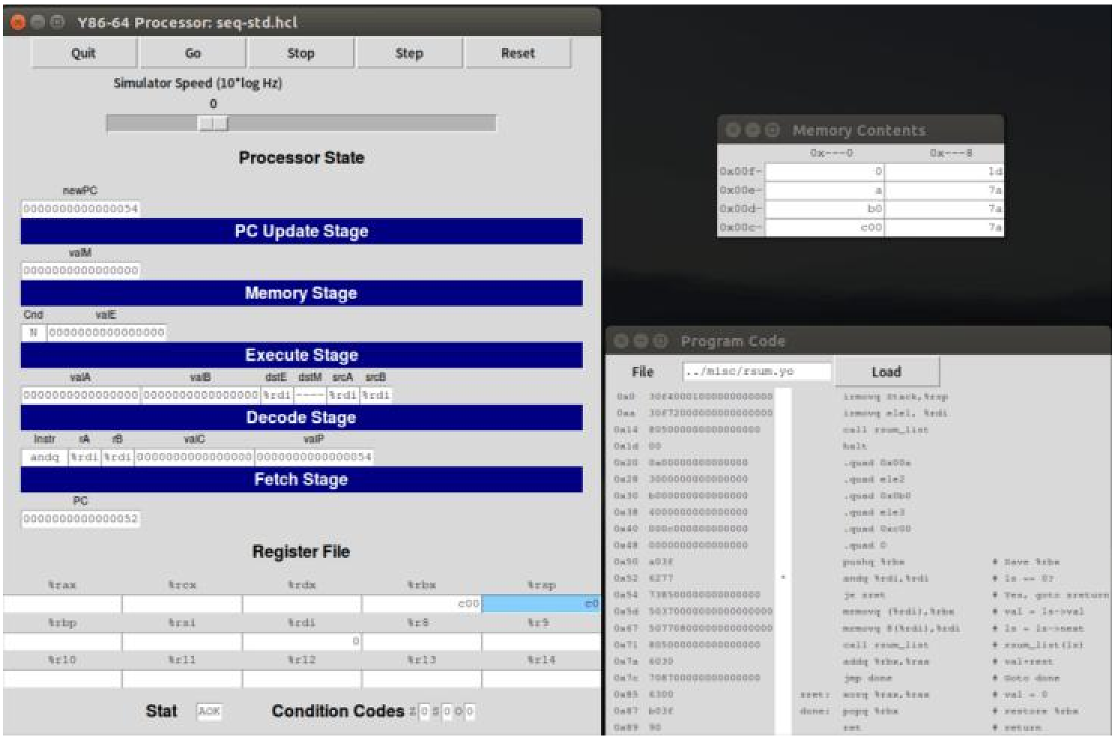
(21) (Third recursion) The rsum_list function is called recursively. First, subtract the stack pointer %rsp by 8(1 byte), and then the address 0x7a of the original next execution address is stored to the location pointed by %rsp. PC is updated to the starting address of rsum_list function.



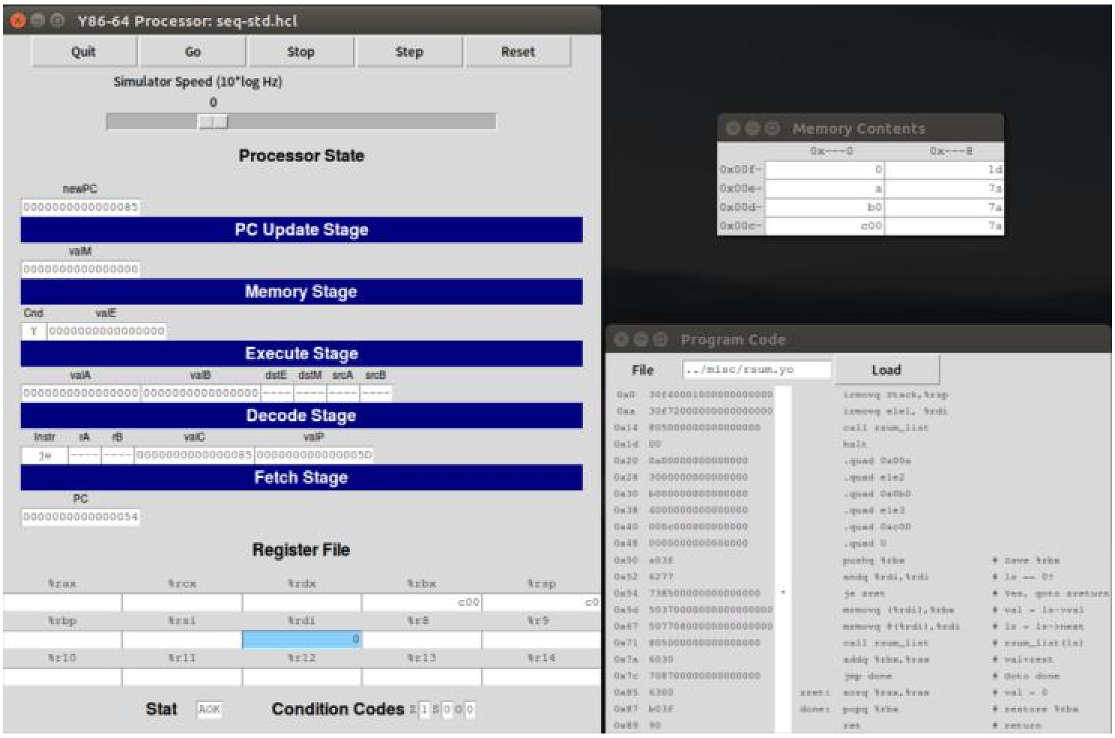
(22) Use the PUSHQ instruction to push the value stored in register %rbx. And in the current step, %rbx has stored the value 0xc00 of the first node in the linked list stored when rsum_list was last called, hence, the stack pointer %rsp will be subtracted by 8 and store 0xc00 in where it points to.



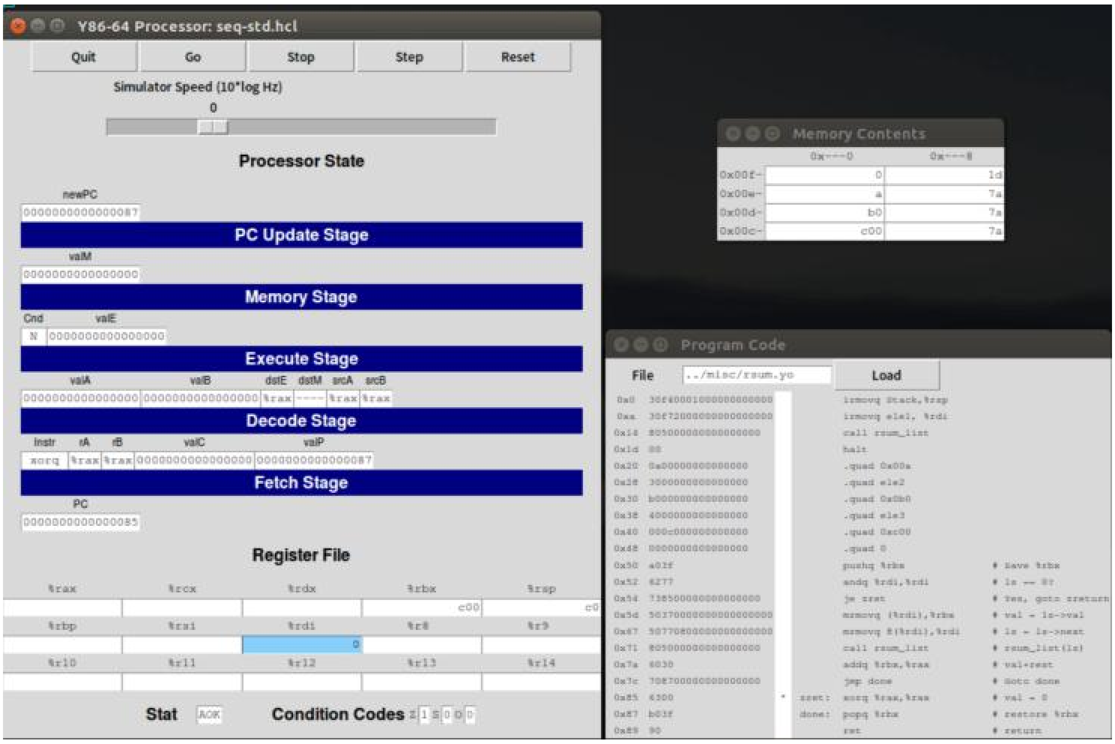
(23) Again, use the ANDQ instruction to check whether the value of %rdi is NULL (0), if %rdi is 0 and the result of the ANDQ instruction is 0 (NULL), the condition code ZF will be set to 1. This time the result of ANDQ instruction is 0.



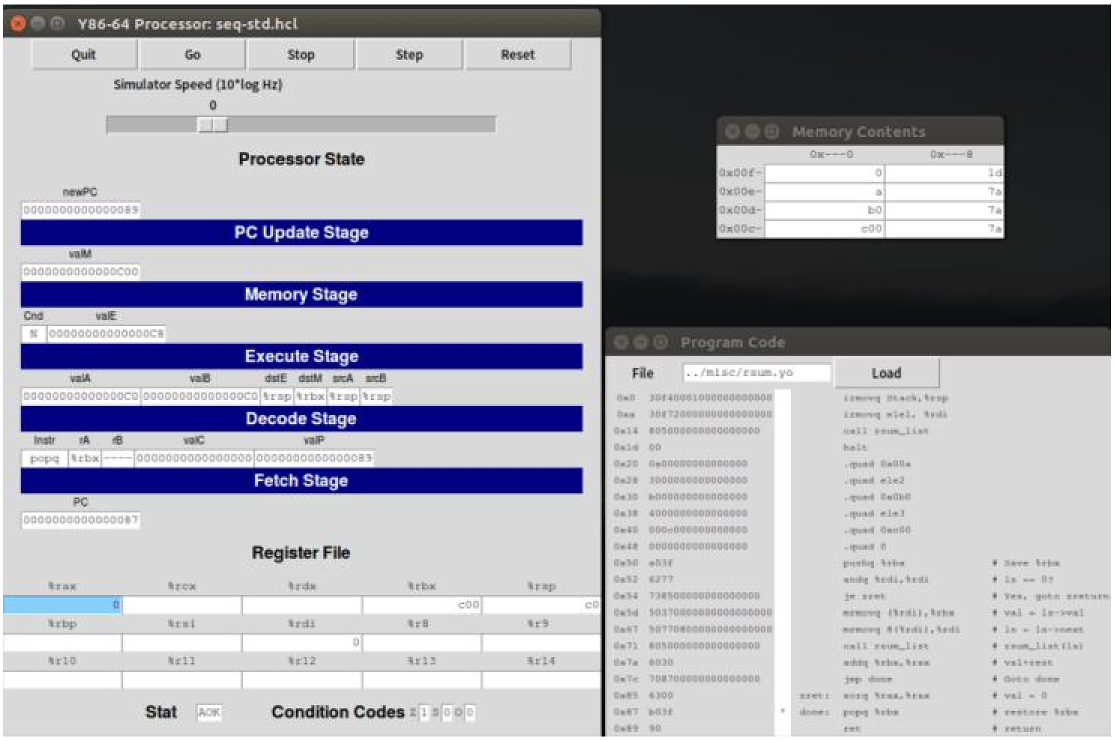
(24) The JE instruction is used again to determine whether to jump according to the condition code ZF set in the previous ANDQ instruction. The current register %rdi is 0, ZF is 1, and Cnd is Y. Therefore, the contents of JE instruction will be executed, and the PC is updated to the address of label zret pointed by JE.



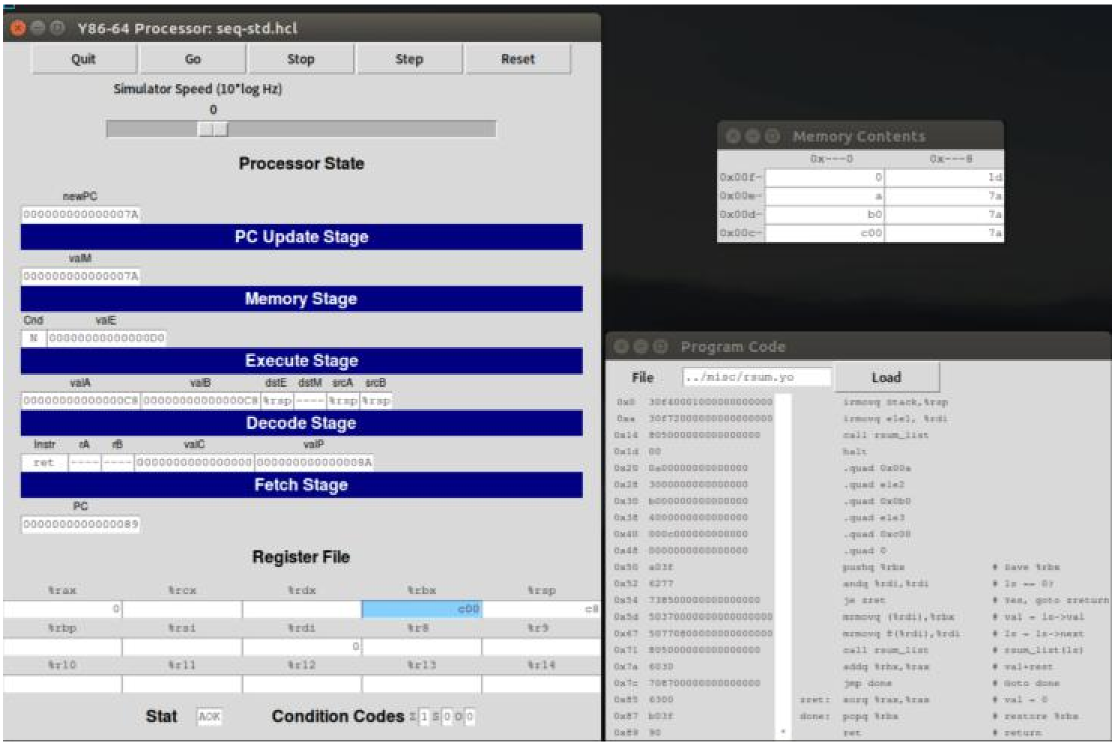
(25) When reaching the label zret and execute the XORQ instruction on the register %rax itself, setting the value of %rax to 0. This is a preparation step since the recursive function reaches the recursive boundary, where a value of 0 is prepared to returne to the recursive function in the previous layer.



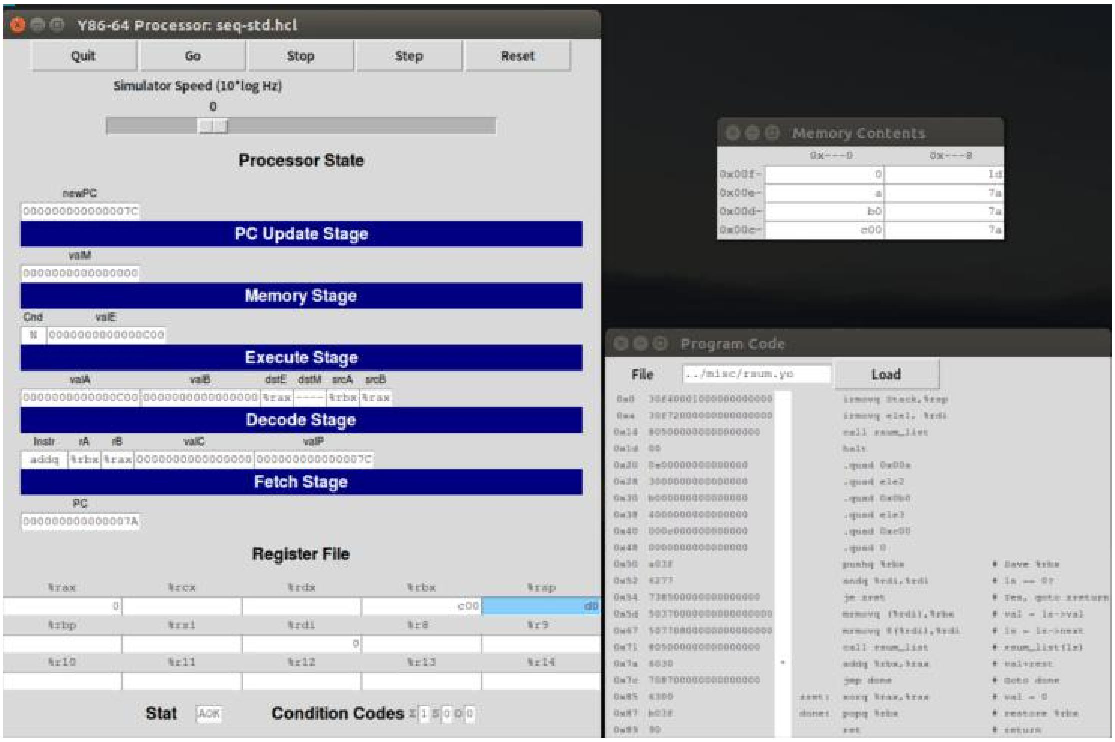
(26) The POPQ instruction is used to fetch the value saved in the stack pointer %rsp from the last function call and put it into register %rbx (0xc00). At the same time, the value of %rsp + 8 (1 byte) indicates that this part of the stack has been poped.



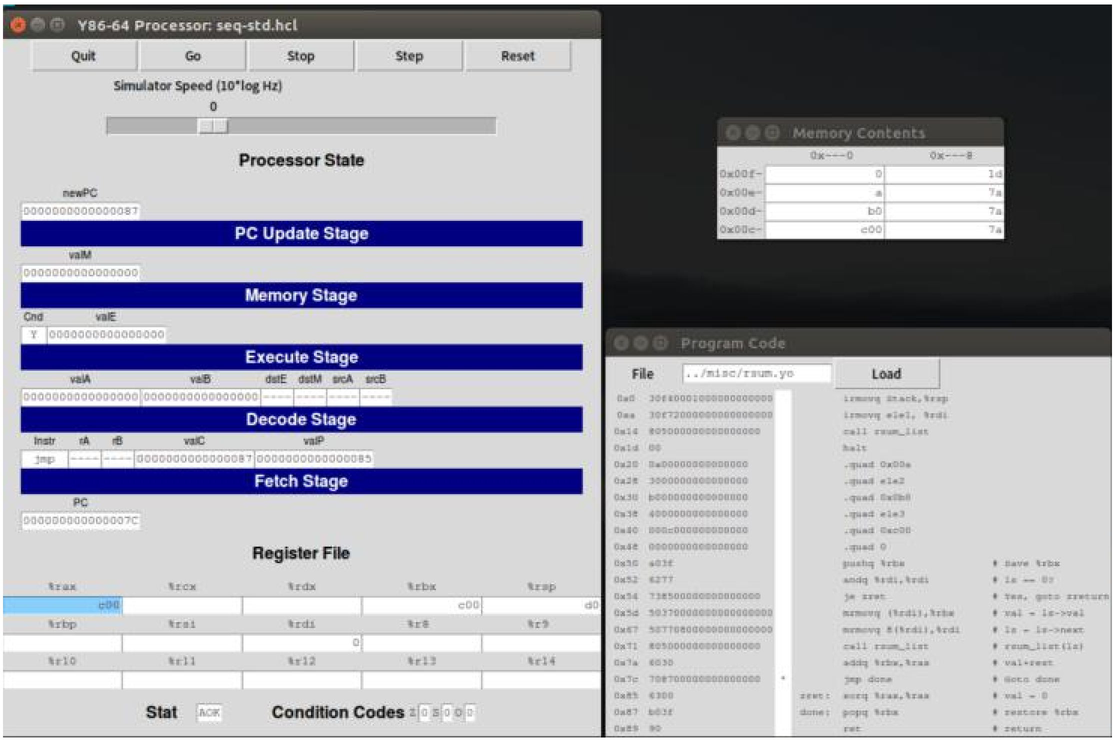
(27) Execute the RET instruction, representing the end of function execution, return the execution result saved in register %rax(0x0), PC will be updated with the instruction address (0x7a) of the stack when the CALL instruction was issued last time, and the stack pointer + 8(1 byte) indicates that the stack element is poped.



(28) Back to the previous recursive function, add the value in %rbx to %rax using the ADDQ instruction.

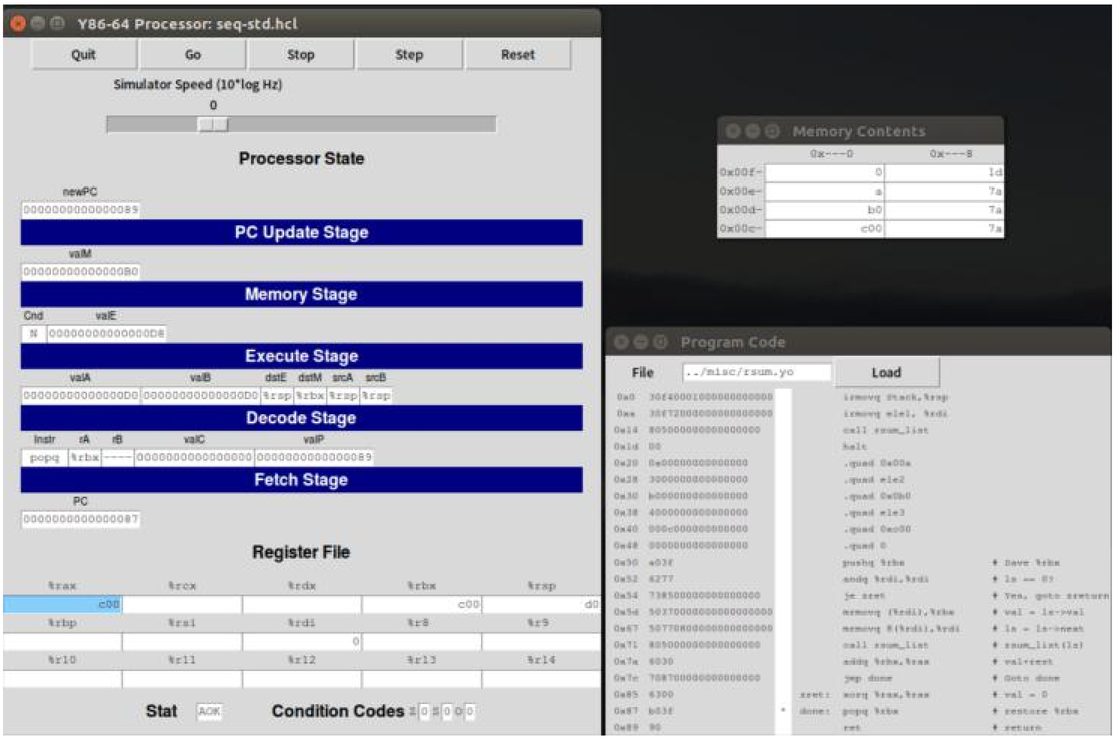


(29) JMP instruction is an unconditional jump instruction. Cnd is directly set to Y, and PC is updated to the instruction of label done pointed by JMP.

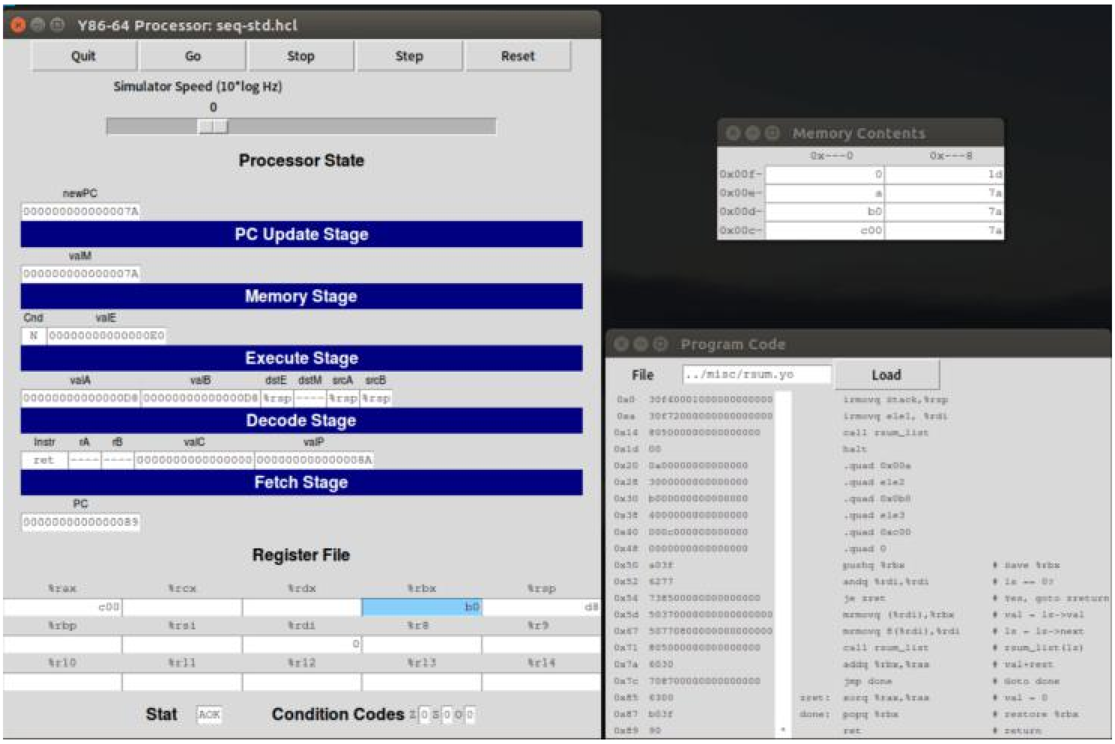


(30) The POPQ instruction is used to fetch the value saved in the

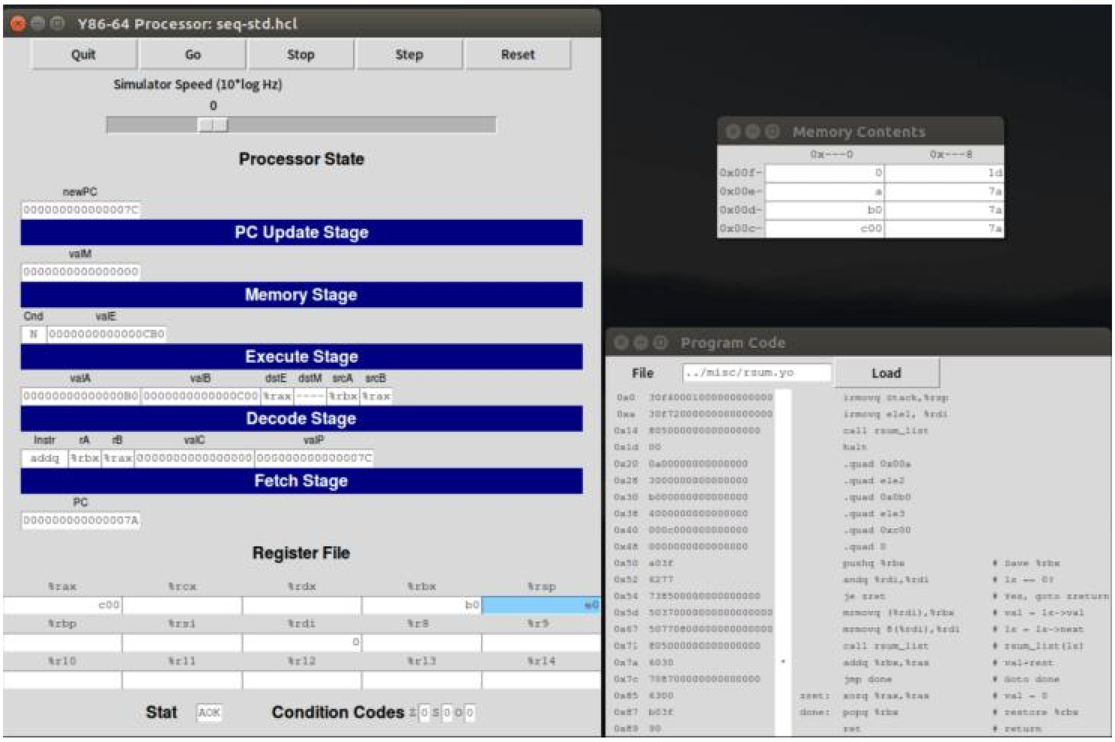
stack pointer %rsp from the last function call and put it into register %rbx (0xb0). At the same time, the value of %rsp + 8 (1 byte) indicates that this part of the stack has been poped.



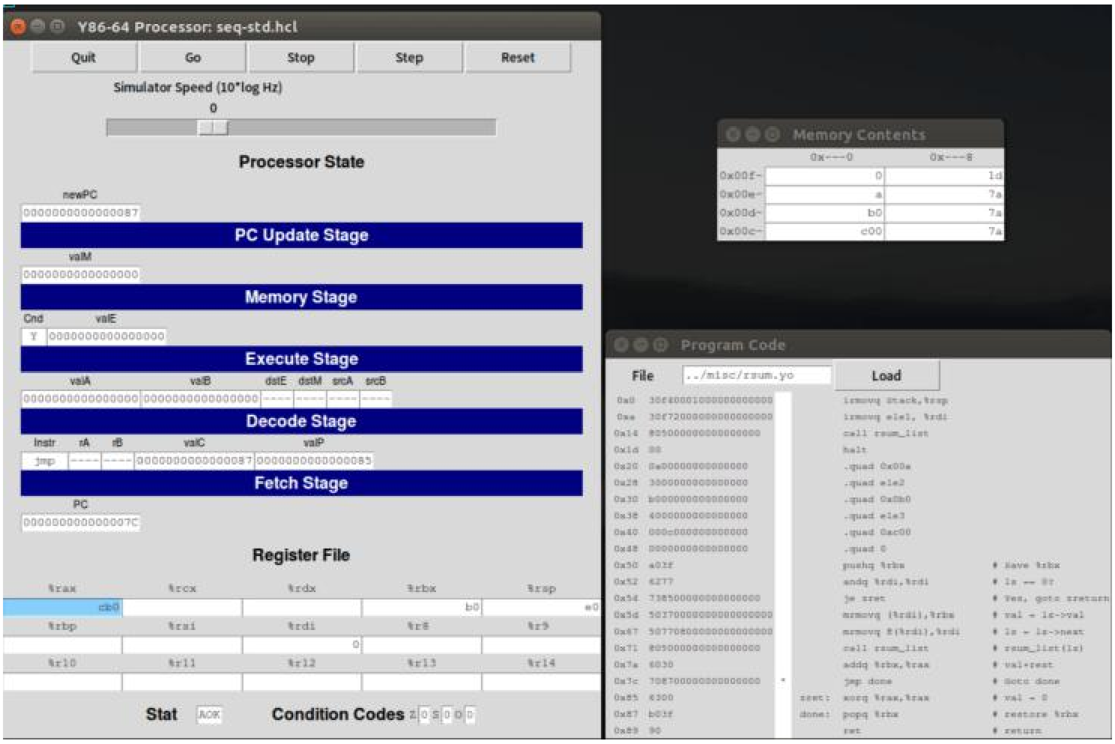
(31) Execute the RET instruction, representing the end of function execution, return the execution result saved in register %rax(0xc00), PC will be updated with the instruction address (0x7a) of the stack when the CALL instruction was issued last time, and the stack pointer + 8(1 byte) indicates that the stack element is poped.



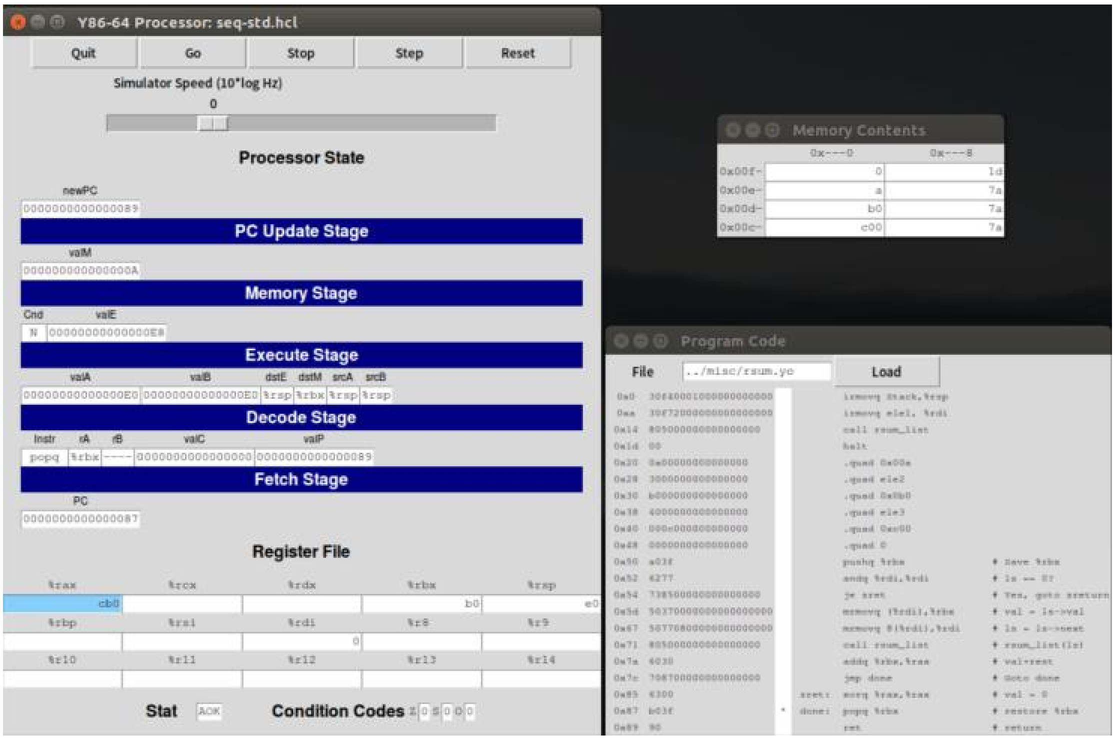
(32) Back to the previous recursive function, add the value in %rbx to %rax using the ADDQ instruction.



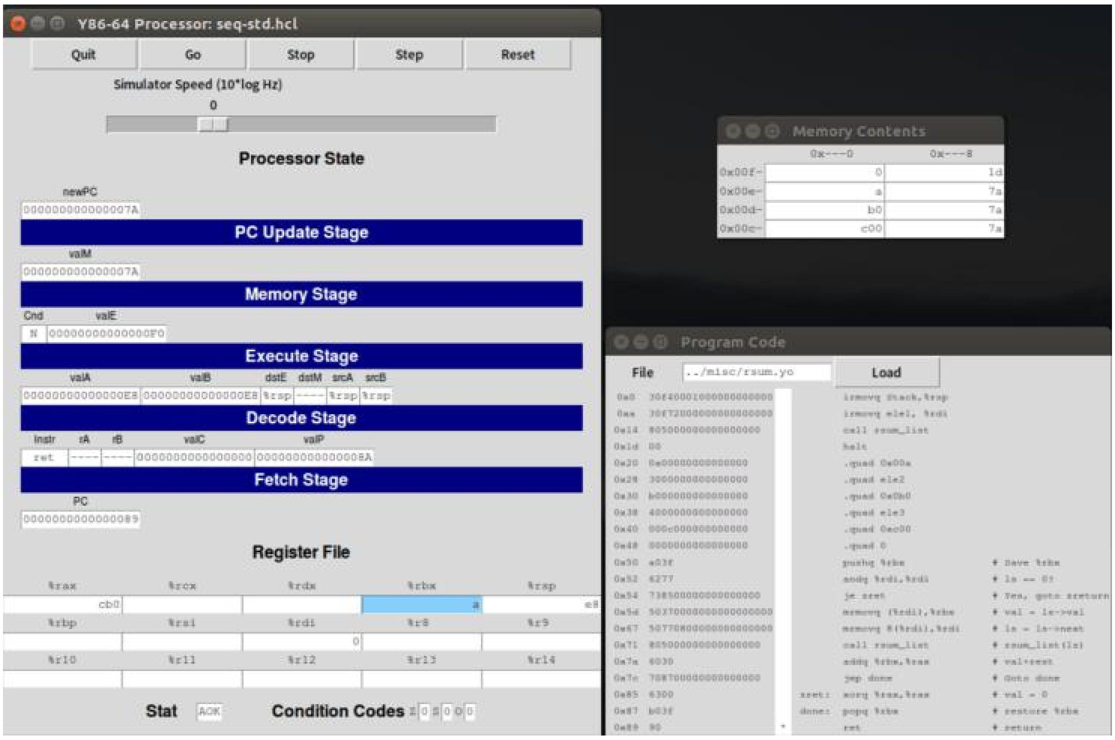
(33) Use JMP instruction, Cnd is directly set to Y, and PC is updated to the instruction of label done pointed by JMP.



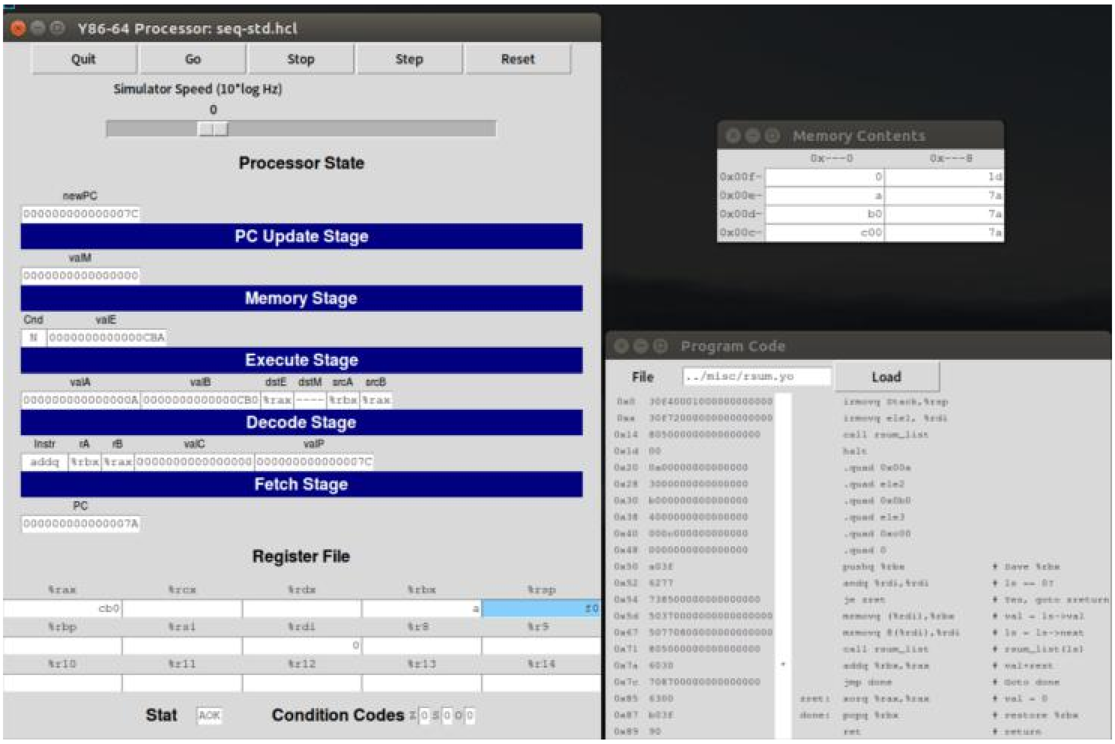
(34) The POPQ instruction is used to fetch the value saved in the stack pointer %rsp from the last function call and put it into register %rbx (0xa). At the same time, the value of %rsp + 8 (1 byte) indicates that this part of the stack has been poped.



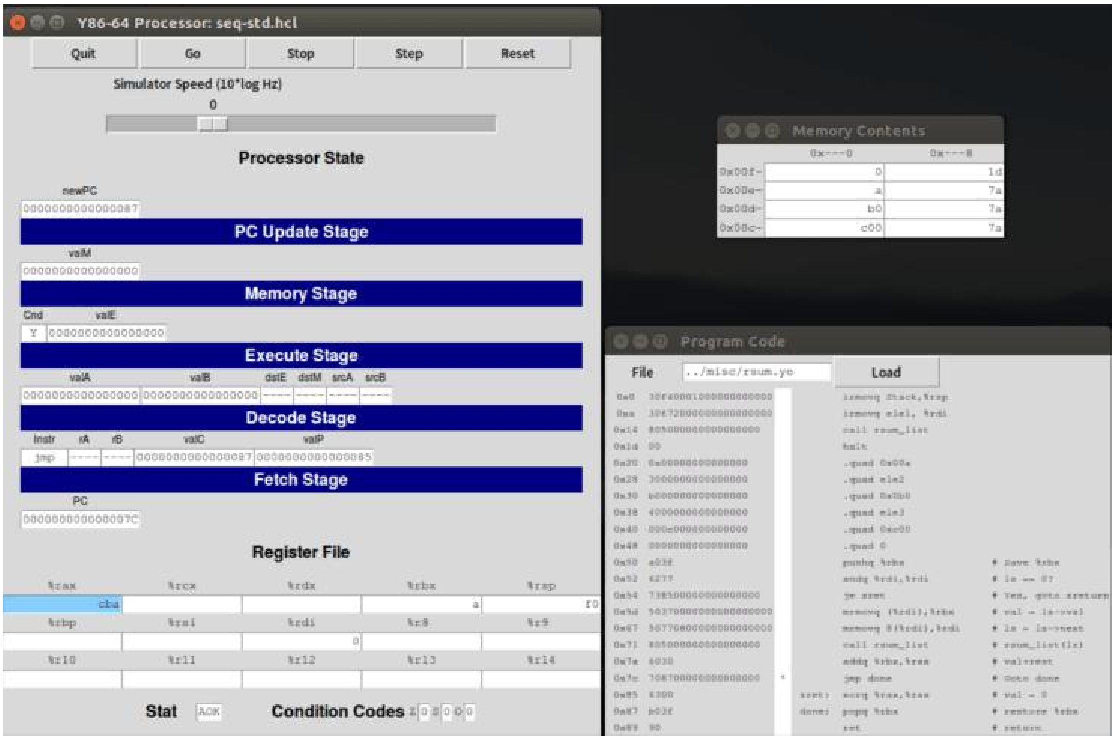
(35) Execute the RET instruction, representing the end of function execution, return the execution result saved in register %rax(0xcb0), PC will be updated with the instruction address (0x7a) of the stack when the CALL instruction was issued last time, and the stack pointer + 8(1 byte) indicates that the stack element is poped.



(36) Back to the previous recursive function, add the value in %rbx to %rax using the ADDQ instruction.

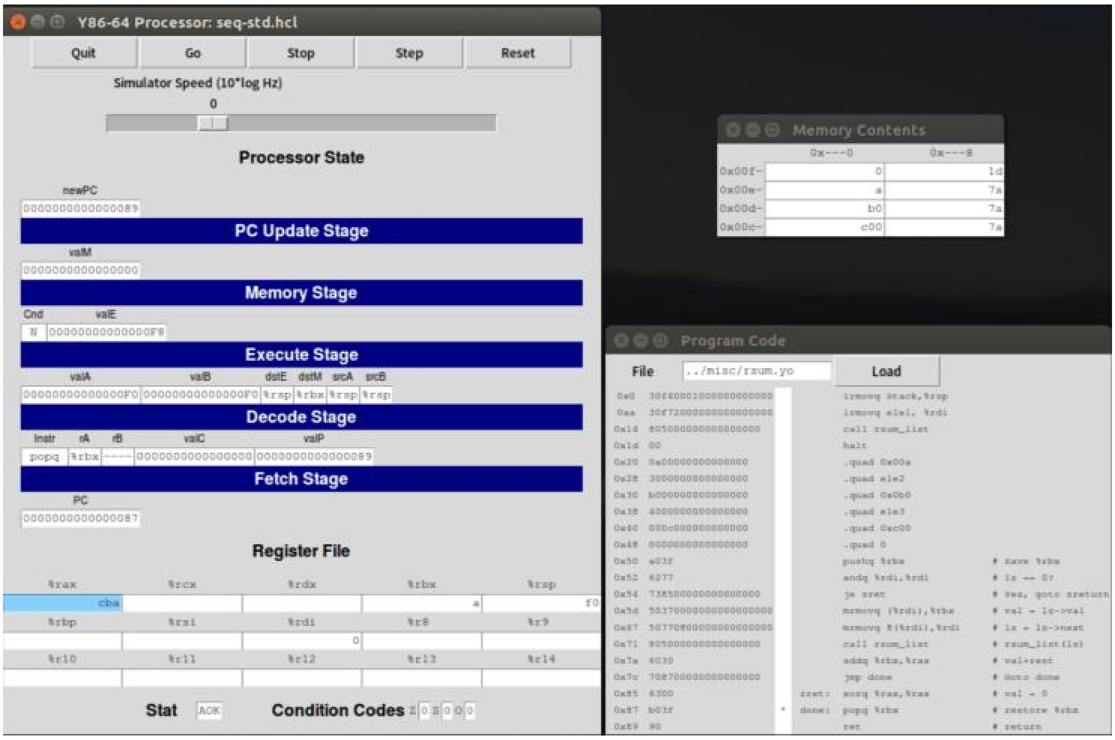


(37) Use JMP instruction, Cnd is directly set to Y, and PC is updated to the instruction of label done pointed by JMP.

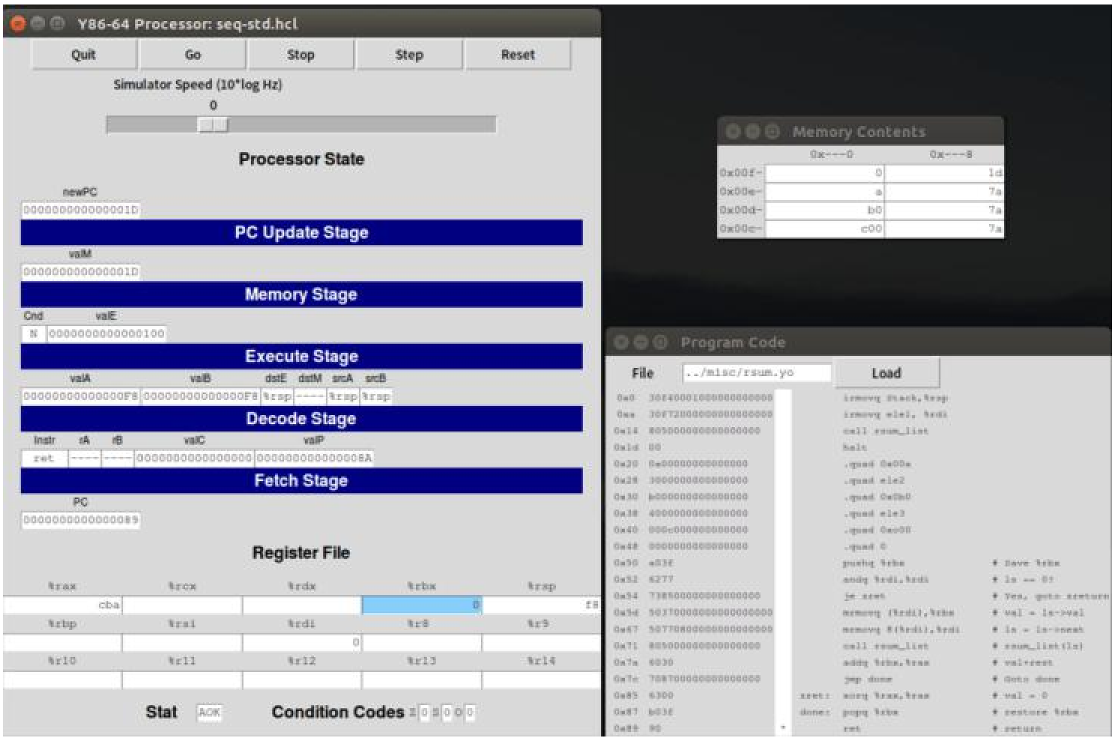


(38) The POPQ instruction is used to fetch the value saved in the stack pointer %rsp from the last function call and put it into register

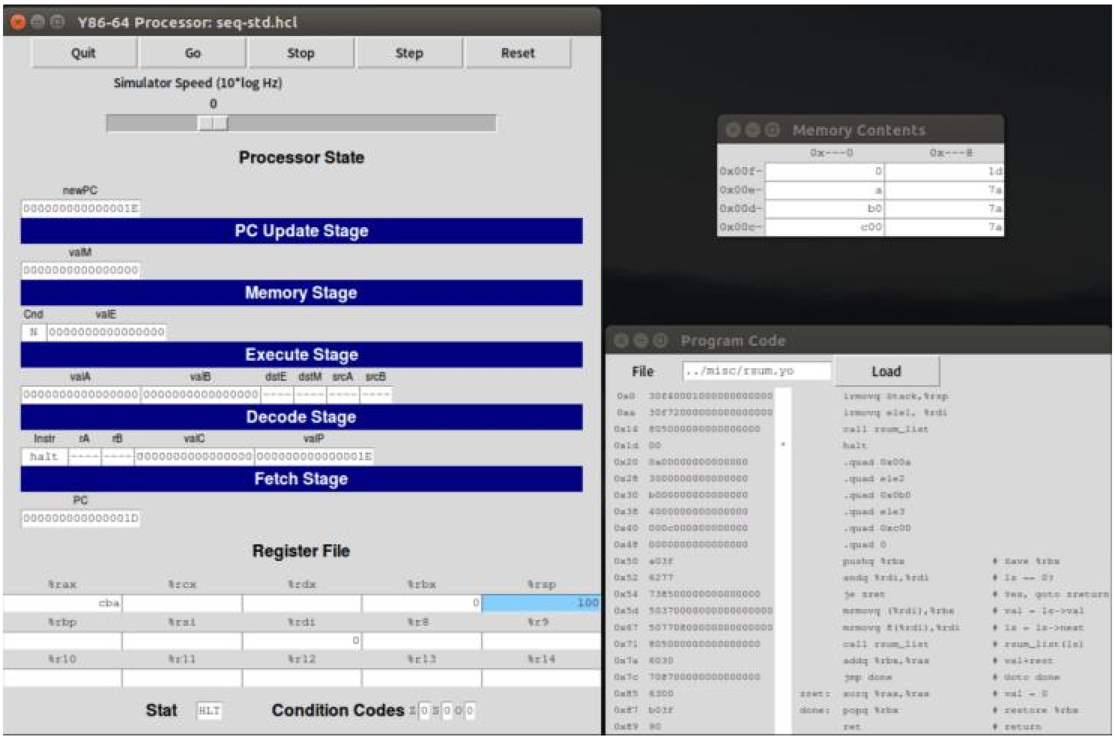
%rbx (0x0). At the same time, the value of %rsp + 8 (1 byte) indicates that this part of the stack has been poped.



(39) Execute the RET instruction, representing the end of function execution, return the execution result saved in register %rax(0xcba), PC will be updated with the instruction address (0x1d) of the stack when the CALL instruction was issued last time, and the stack pointer + 8(1 byte) indicates that the stack element is poped.



(40) Jump to 0x1d, execute HALT instruction, change the execution status of the program from AOK to HLT, indicating that the program stops running.



5. Appendix (Program Code)

- sum.ys

```
# Initial code
 irmovq Stack, %rsp
 irmovq ele1, %rdi
 call sum_list
 halt

# Sample linked list
.align 8
ele1:
 .quad 0x00a
 .quad ele2
ele2:
 .quad 0x0b0
 .quad ele3
ele3:
 .quad 0xc00
 .quad 0

# long sum_list(list_ptr ls)
# ls in %rdi
sum_list:
 xorq %rax, %rax          # val = 0
 andq %rdi, %rdi  # ls == 0?
 je done    # Yes, goto done
loop: mrmovq (%rdi), %r10 # t = ls->val
 mrmovq 8(%rdi), %rdi # ls = ls->next
 addq %r10, %rax  # val += t
 andq %rdi, %rdi  # Check ls
 jne loop         # If null, goto done
done: ret    # return

.pos 0x100
Stack:
```

- rsum.ys

```
# Initial code
 irmovq Stack, %rsp
 irmovq ele1, %rdi
 call rsum_list
 halt
```

```
# Sample linked list
.align 8
ele1:
 .quad 0x00a
 .quad ele2
ele2:
 .quad 0x0b0
 .quad ele3
ele3:
 .quad 0xc00
 .quad 0

# long rsum_list(list_ptr ls)
# ls in %rdi
rsum_list:
 pushq %rbx   # Save %rbx
 andq %rdi, %rdi   # ls == 0?
 je zret    # Yes, goto zreturn
 mrmovq (%rdi), %rbx # val = ls->val
 mrmovq 8(%rdi), %rdi # ls = ls->next
 call rsum_list  # rsum_list(ls)
 addq %rbx, %rax   # val+rest
 jmp done   # Goto done
zret: xorq %rax, %rax   # val = 0
done: popq %rbx   # restore %rbx
 ret    # return

.pos 0x100
Stack:
```