## Lab 3: Simulating Y86-64 Program

## 1 Introduction

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

## 2 Lab Instructions or Steps

**1.Source Changing and install**

1.1 Find Sources List:

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

For Ubuntu 16.04:

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

1.2 Backup: sources.list.bak

|  |
| --- |
| sudo cp /etc/apt/sources.list /etc/apt/sources.list.bak |

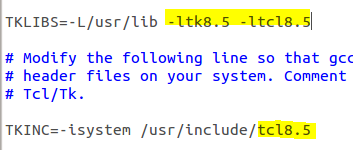
1.3 Modify: Rewrite sources.list using source from the website.

|  |
| --- |
| sudo gedit /etc/apt/sources.list |

1.4 Update:

|  |
| --- |
| sudo apt-get update (upgrade / -f install)  sudo apt-get install flex bison  sudo apt-get install tcl8.5 tcl8.5-dev tk8.5 tk8.5-dev |

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



**2.Doing experiment**

2.1Start by copying the Lab3 to a directory in which you plan to do your work.

2.1.1Unpacked 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 by all 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.



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

2.2 Change to the sim directory and build the Y86-64 tools:

|  |
| --- |
| make clean  make |

Note: For the Y86-64processor, 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 automatedtesting. 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 ofthe 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).

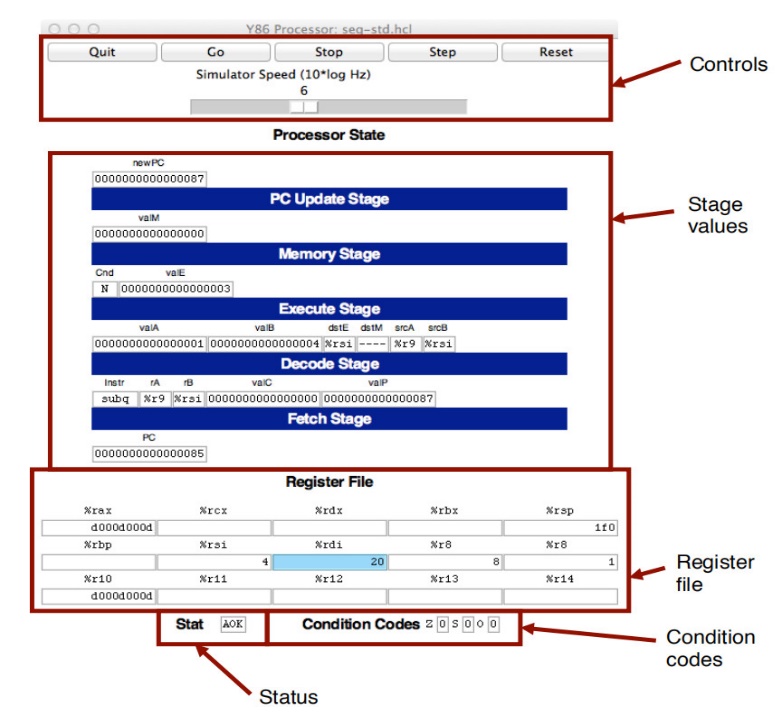
-v n (TTY mode only) Sets the verbosity level to n, which must be between 0 and 2 with a default valueof 2.

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

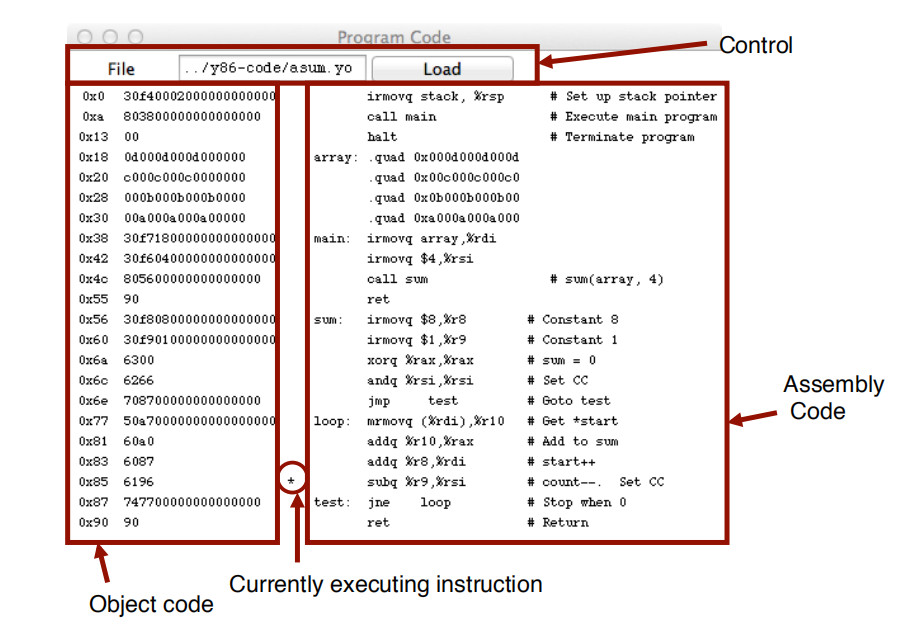
2.2.1 In seq directory, use following command to open GUI:

|  |
| --- |
| make clean  make ssim VERSION=std  ./ssim -g ../y86-code/asum.yo |

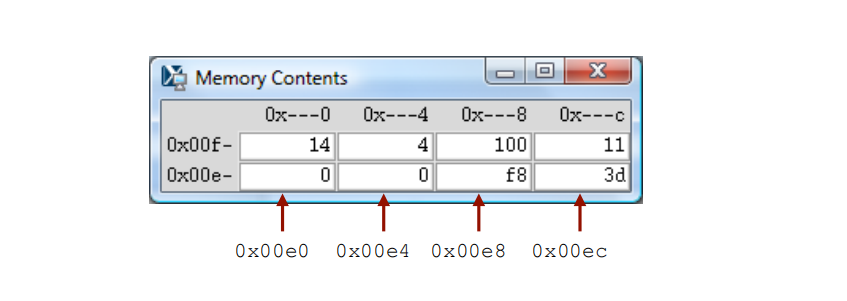
This code runs SSIM in GUI mode, executing the instructions in object code file *asum.yo* from the y86-code subdirectory.



*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 simulatort oexecuteoneinstructionand 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 differ ence is that the simulator displays the name of the instruction in a field labeled Instr, 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 nonzero 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 theminimum and maximum addresses that have changed since the program began executing. Each row showsthe contents of two memory words. Thus, each row shows 16 bytes of the memory, where the addressesof 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 placeso far, we see that *%rsp* was initialized to *0x200* (line 3). The call to *main* on line 4 pushes the returnpointer *0x013*, which is written to address *0x01f8*. Procedure *main* calls *sum* (line 16), causing there turn 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*.

2.3Y86\_Lab

2.3.1You 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*.

**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.

**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.

2.3.2Testing 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 |

2.4 Conclusion

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

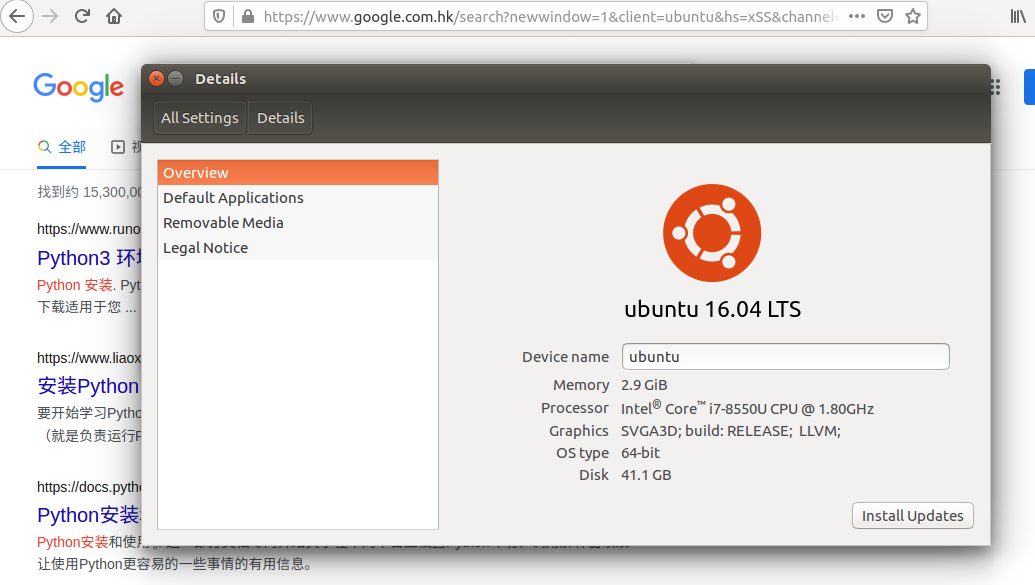
2.4.1 Explain the function and processor state of each step in the source file of sum.ys and rsum.ys according to your snapshot.

## 3 Lab Device and Environment

Device

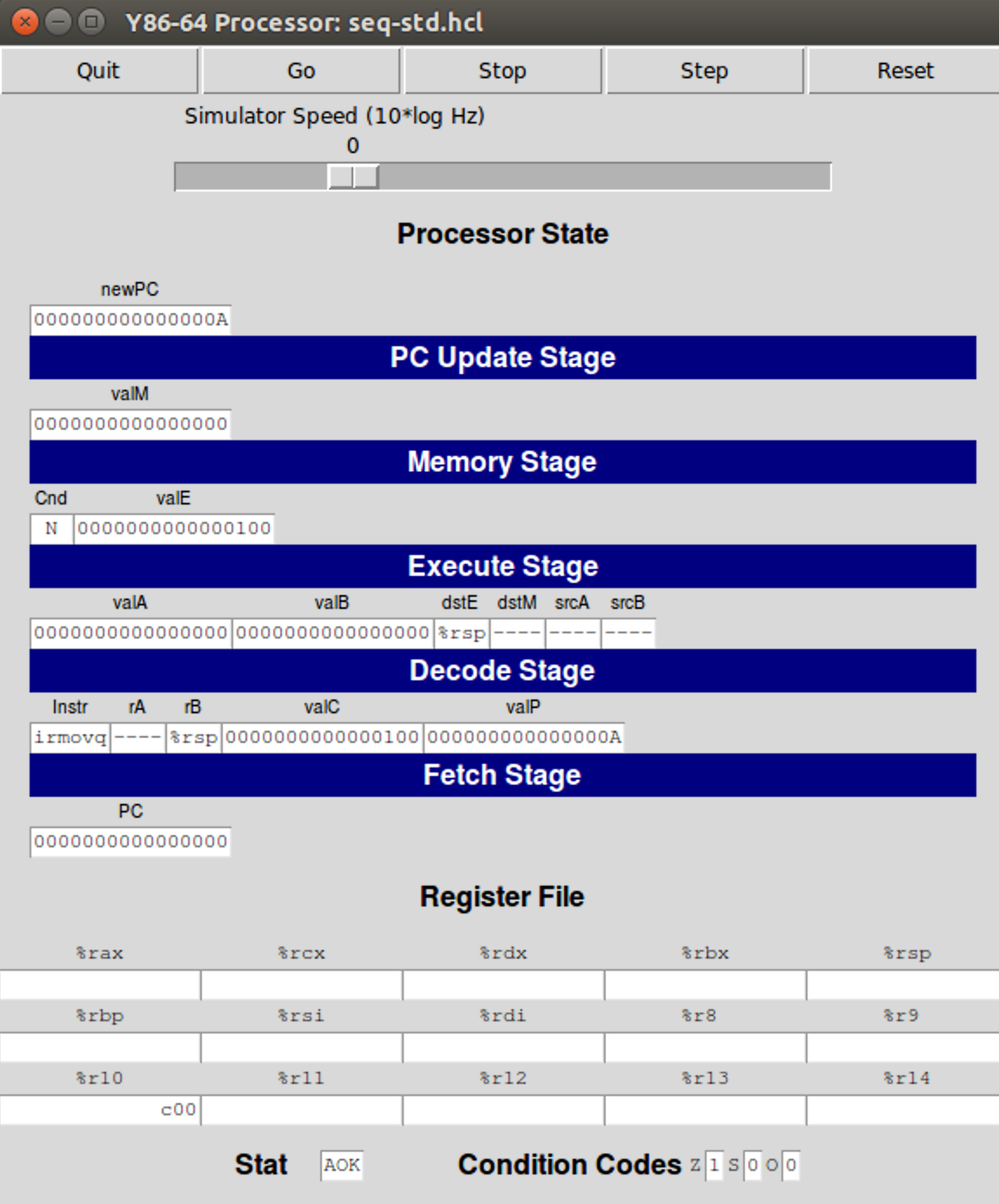
|  |
| --- |
| 版本 Windows 10 家庭版  版本号 21H1  安装日期 ‎2021/‎7/‎30  操作系统内部版本 19043.1320  版本 Windows 10 家庭版  版本号 21H1  安装日期 ‎2021/‎7/‎30  操作系统内部版本 19043.1320 |

Lab Environment



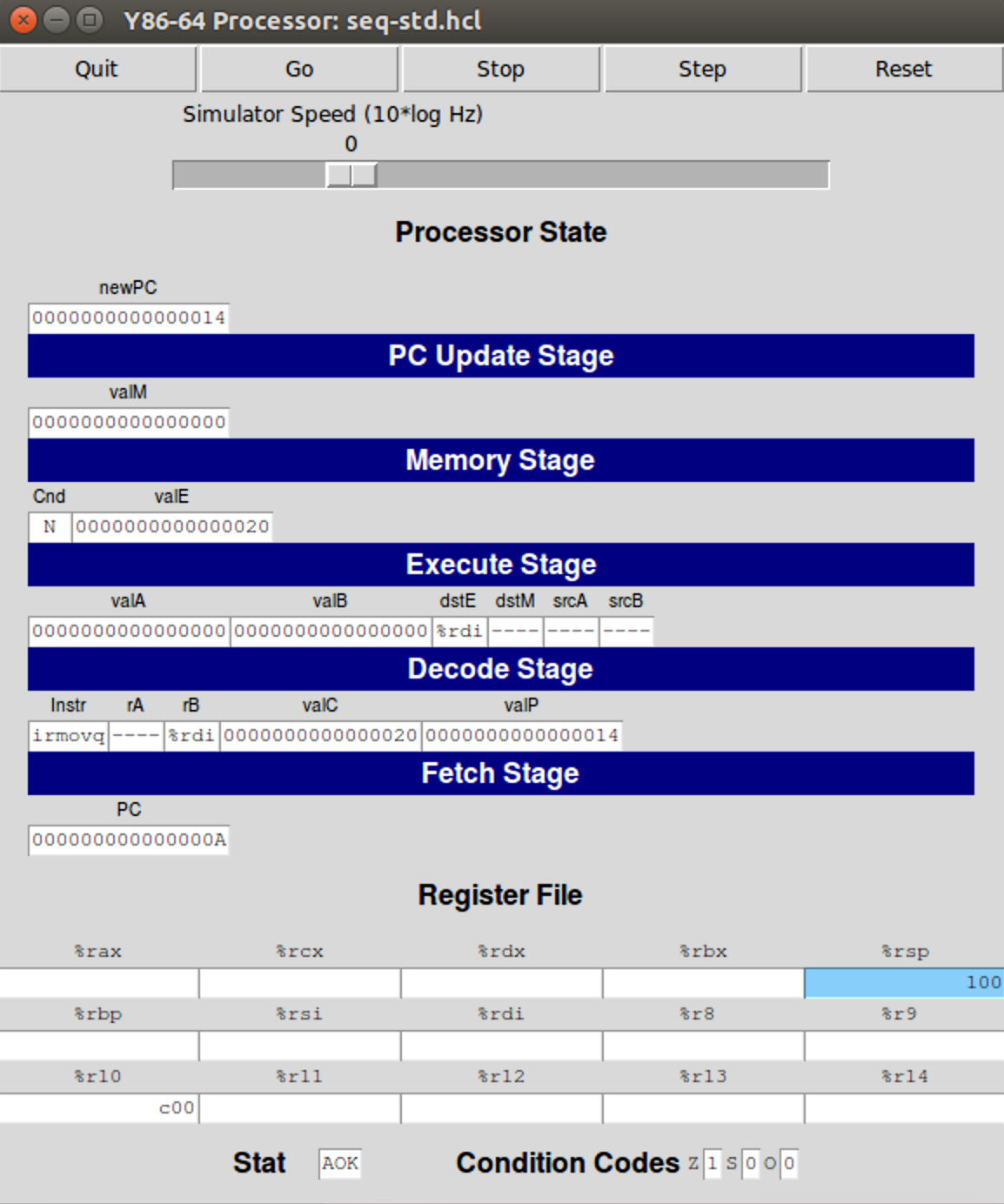
## 4 Results and Analysis

1.The instruction is “nop”, it does nothing.



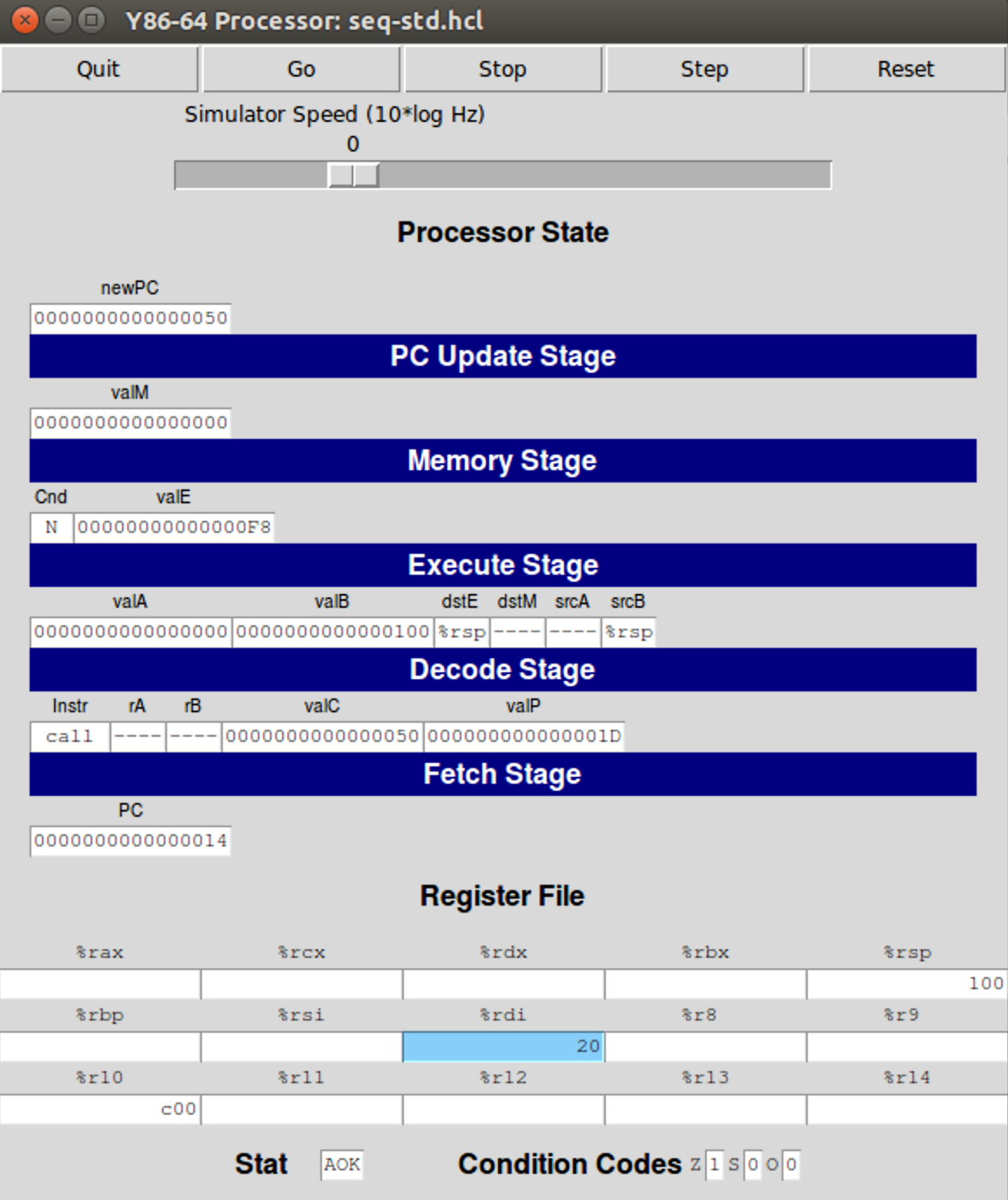
2. irmovq Stack,%rsp

It copies an immediate value (100) to a register (%rsp), where %rsp is a stack pointer.

+

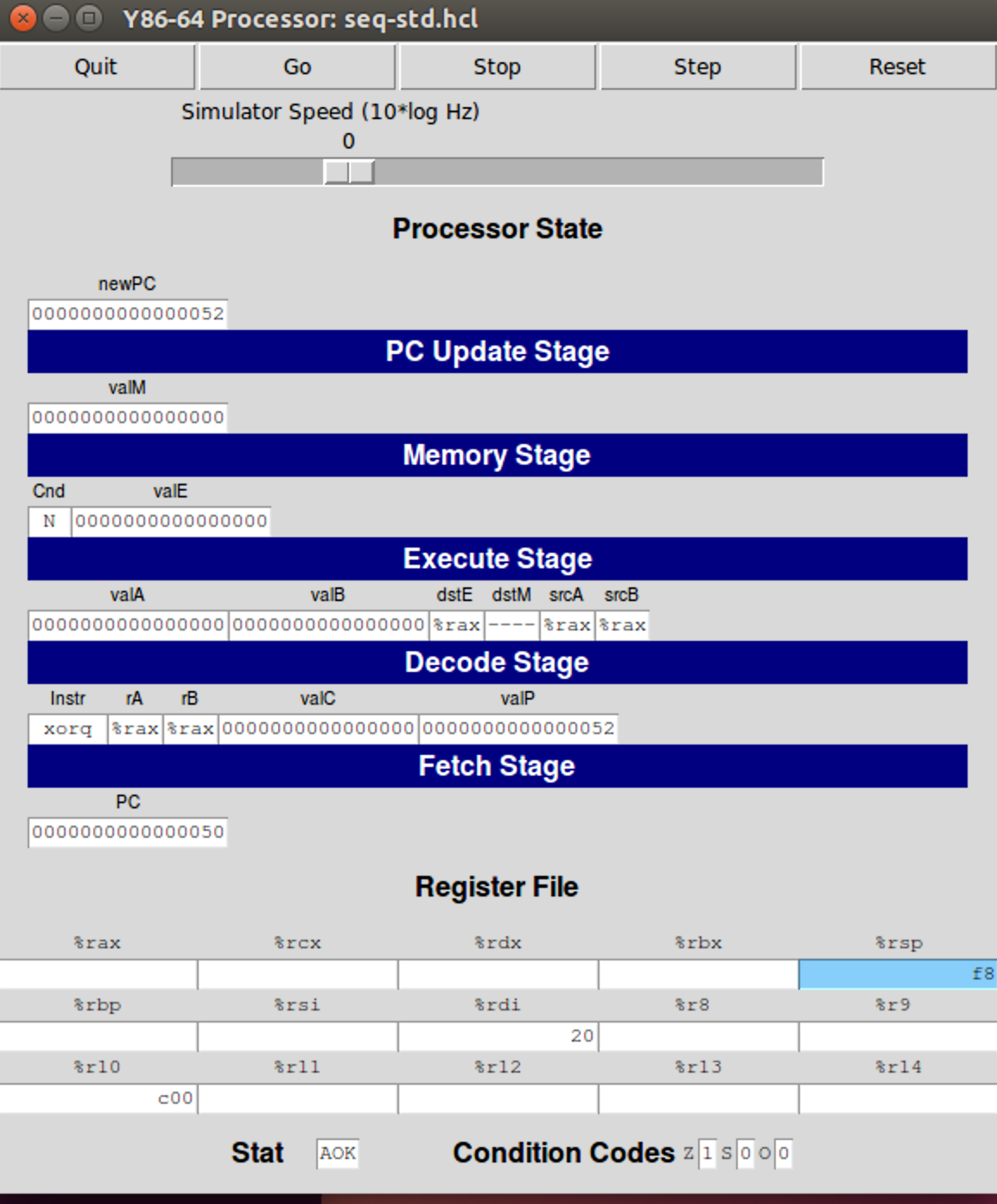
3.irmovq ele1, %rdi.

It copies an immediate value (20) to register %rdi. Note that 20 is where the first node is stored.



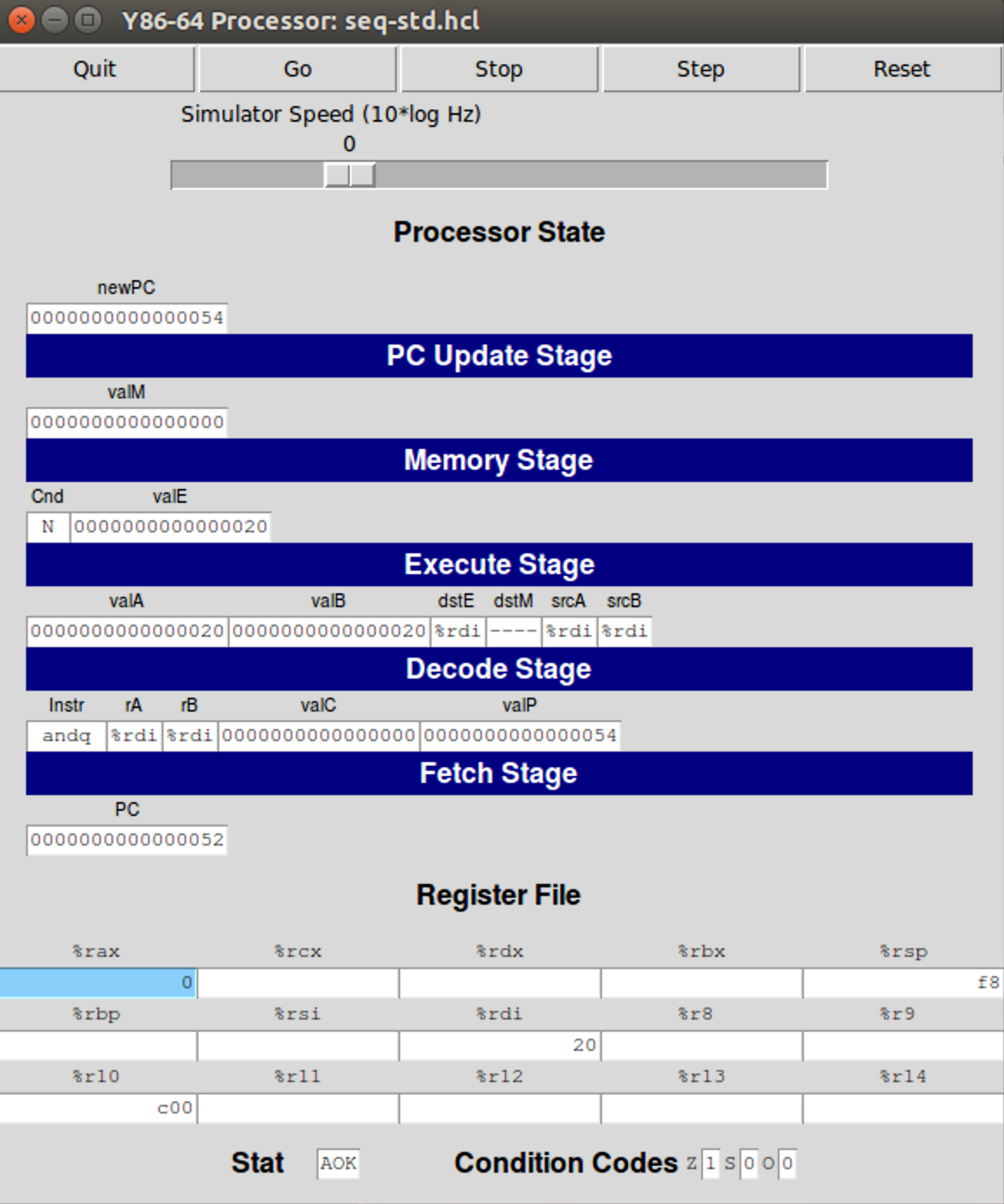
4. call sum\_list

In the program code, it calls the function named “sum\_list”. It will push the return address onto stack, meaning F8 will be at the top of stack (%rsp pointing to F8), and then jump to the destination address.



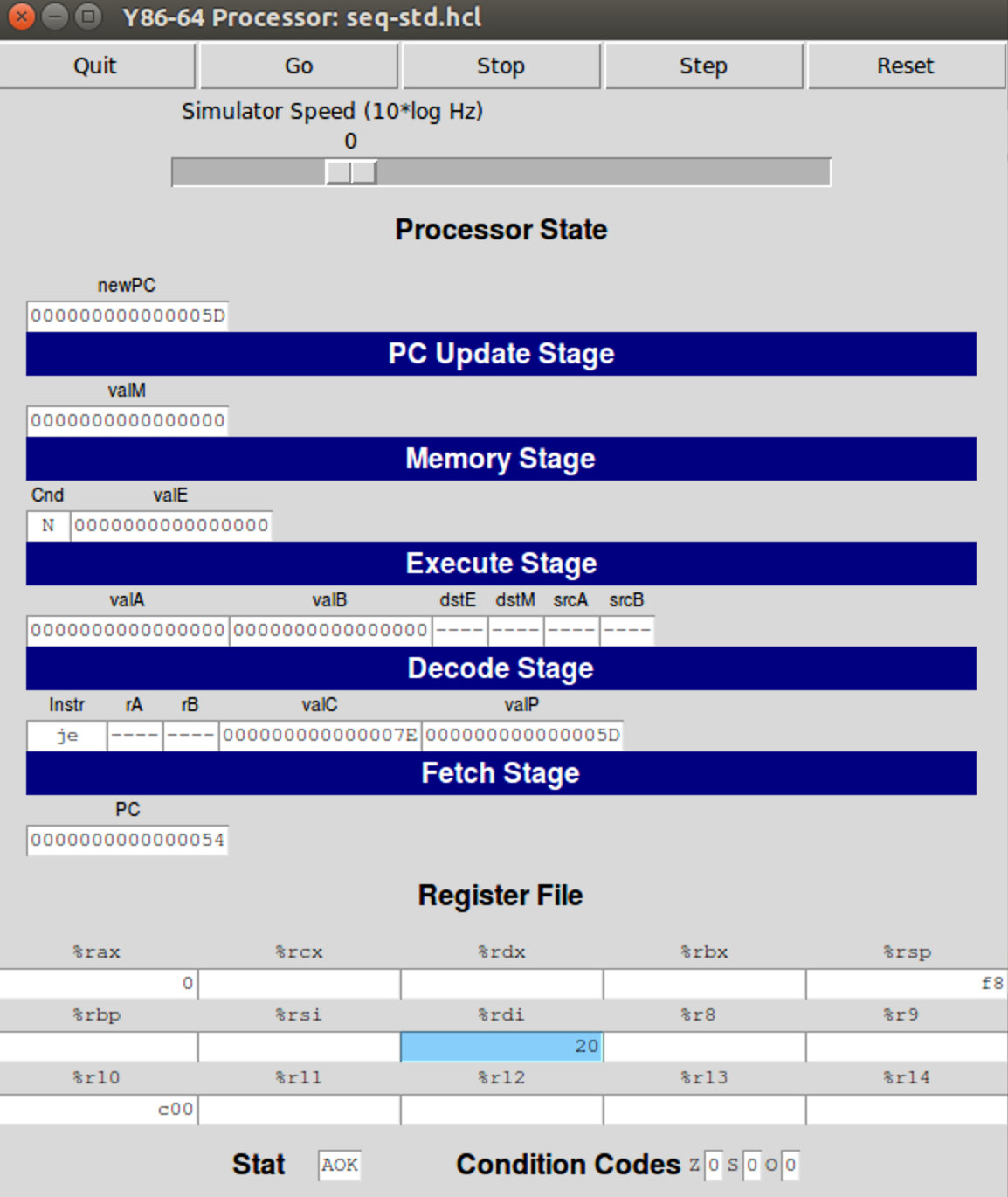
5. xorq %rax,%rax

Exclusive or is true when the compared bits are different. Hence this instruction is set register %rax to 0, which will be returned after whole procedure is done.



6. andq %rdi,%rdi

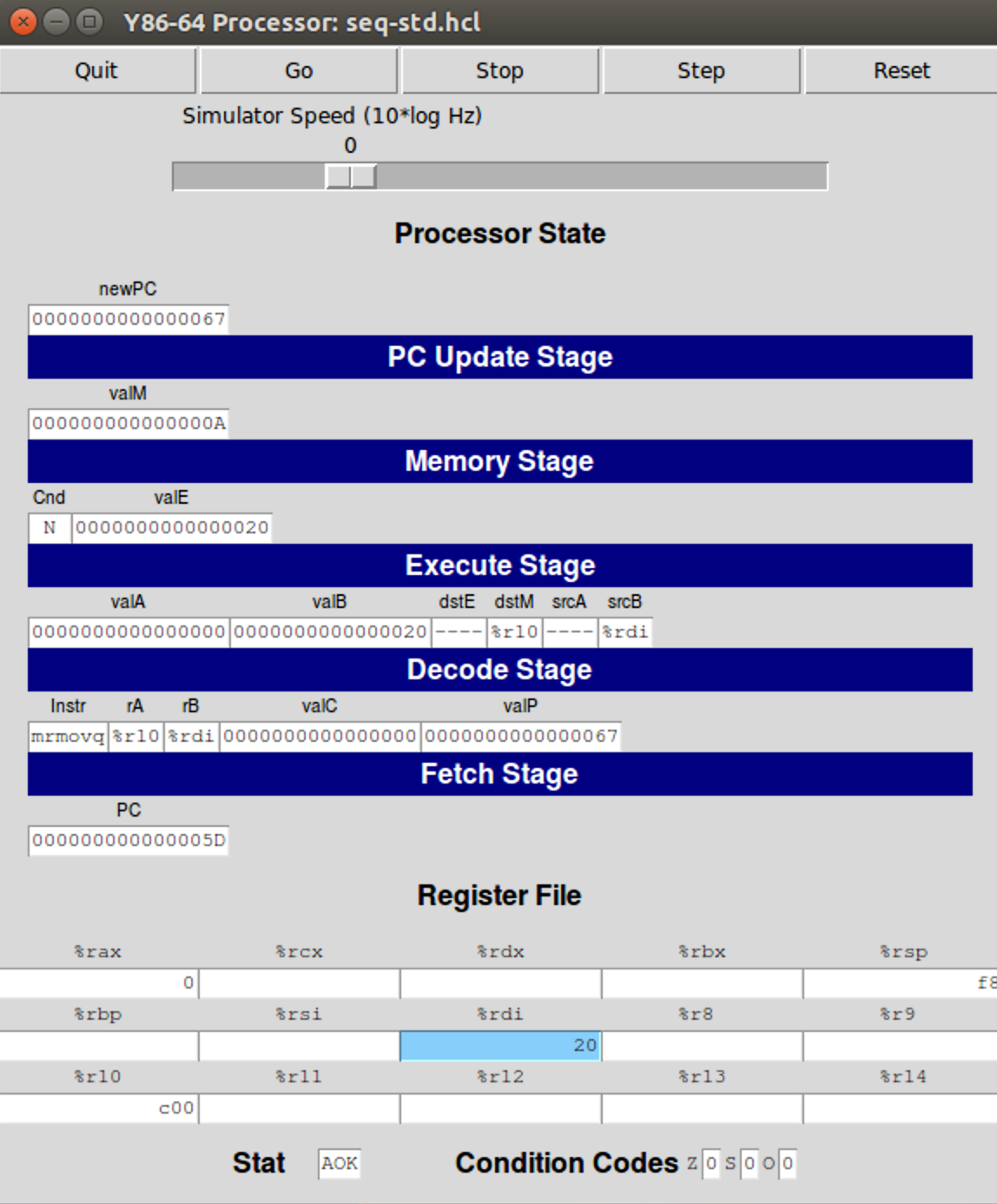
Bitwise and produces 1 if and only if both bits are 1. Hence to do bitswise and with itself yields itself. Since instructions like *cmp* or *test* does not exist in Y86-64 architecture, the purpose of andq here in Y86-64 architecture is to set condition code.



Noting that register %rdi does not hold a zero, making the zero flag a zero as well. The branching is determined by it.

7. je

It will jump to “done” label if executed (when ZF is set). However, the zero flag is 0, meaning equality is not reached. This instruction will not be executed, and now a loop begins.



8. mrmovq (%rdi),%r10

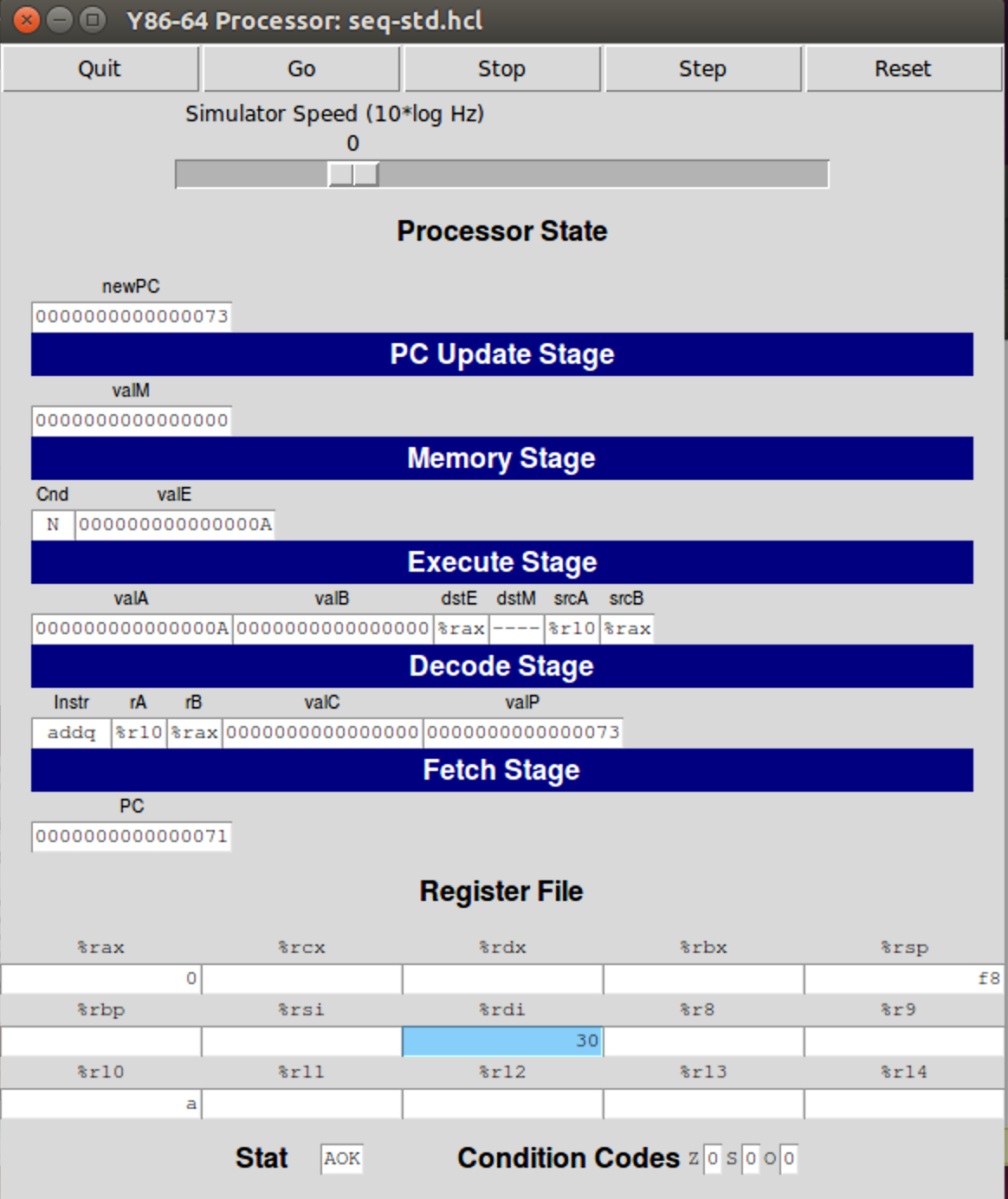
What is stored in %rdi (0x20) will be used directly as an address referencing some memory, then the referenced data, which is the data stored inside the next node, will be copied to register %r10.



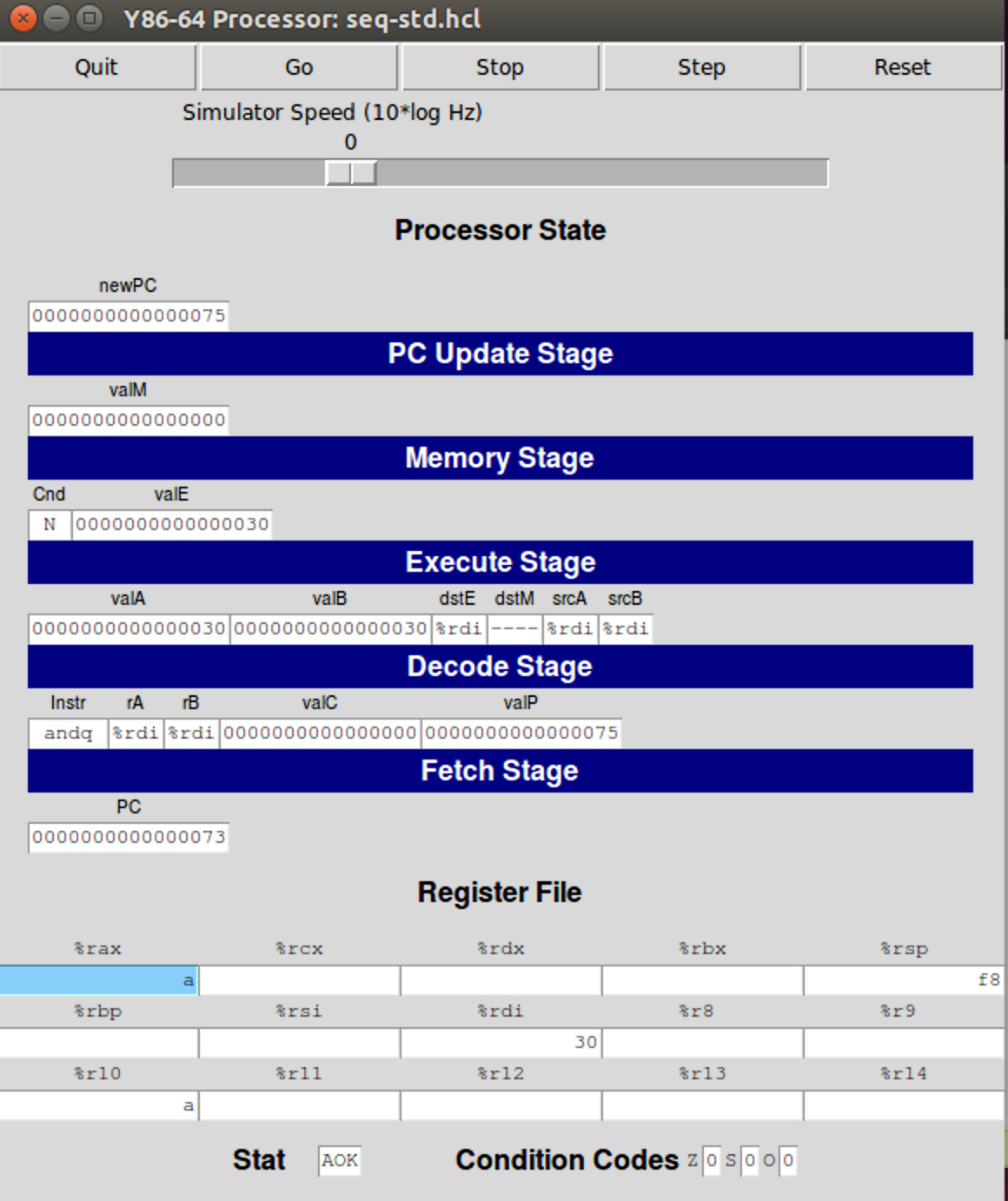
Here we see that the data stored at address 0x20 is “a”, and it is now in register %r10.

9. mrmovq 8(%rdi),%rdi

The complete instruction is mrmovq 8(%rdi), %rdi. The data stored in register %rdi will be used as address and incremented by immediate value of 8. After that the result will be used to dereference some data. Dereferencing 0x28 yields 0x30, the address of the value stored inside the next node. This acknowledgement will be refined by looking at the note appended after this instruction (ls = ls->next) in program code.



Register %rdi is now updated to 30. The 10th instruction is addq %r10, %rax. Data stored in register %r10 and %rax will be added and then stored in %rax. Obviously register %rax will be updated to “a”.



At the end of the loop, the test (or comparison) should be done again to see if the next loop should begin or not. Like mentioned before, andq instruction is used again to do the testing. It not hard to tell that the loop will go again since %rdi holds something not zero.

Inside the loop, at first %rdi has address 0x20, it is dereferenced and the data 0x00a is read. 0x20 is then incremented by 8, now being 0x28 and dereferencing 0x30, hence %rdi is updated to 30 so that the next value can be obtained. This procedure goes on until address 0x48 is dereferenced and %rdi is updated to a null pointer, which will later terminate the loop.

The function is used to sum up the data stored in a linked list using a while loop.

Now we talk about rsum.ys

1.irmovq Stack,%rsp

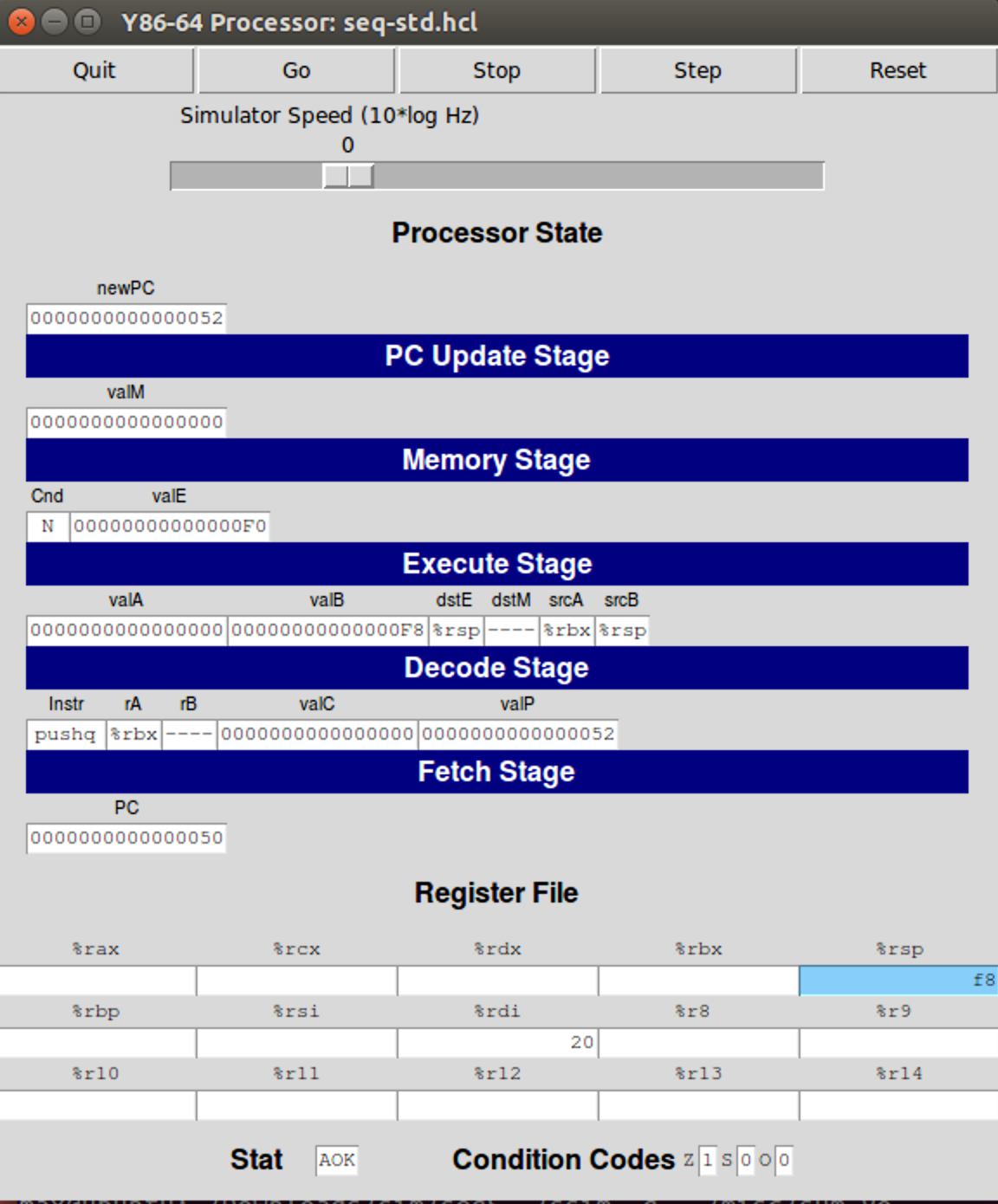
sets up a stack pointer.

2.irmovq ele1,%rdi

copies the address of the first node to register %rdi.

3. call rsum\_list

calls the function named rsum\_list. Function address is pushed onto stack, making register %rsp hold value f8.



4. pushq %rbx.

It causes the stack pointer to decrement by size of a quad word, the value stored in register %rbx will be stored in the new decremented address. Here %rsp will be f0, and since %rbx holds nothing, nothing will be stored at f0.

5. andq %rdi,%rdi

It will set the condition code. Register is not zero, thus the zero flag will be 0.

6.je function

7.mrmovq (%rdi),%rbx.

The value in %rdi (20) will be used as an address to reference some data. 0x00a is stored at 0x20, which will be copied to register %rbx.

8.mrmovq 8(%rdi),%rdi.

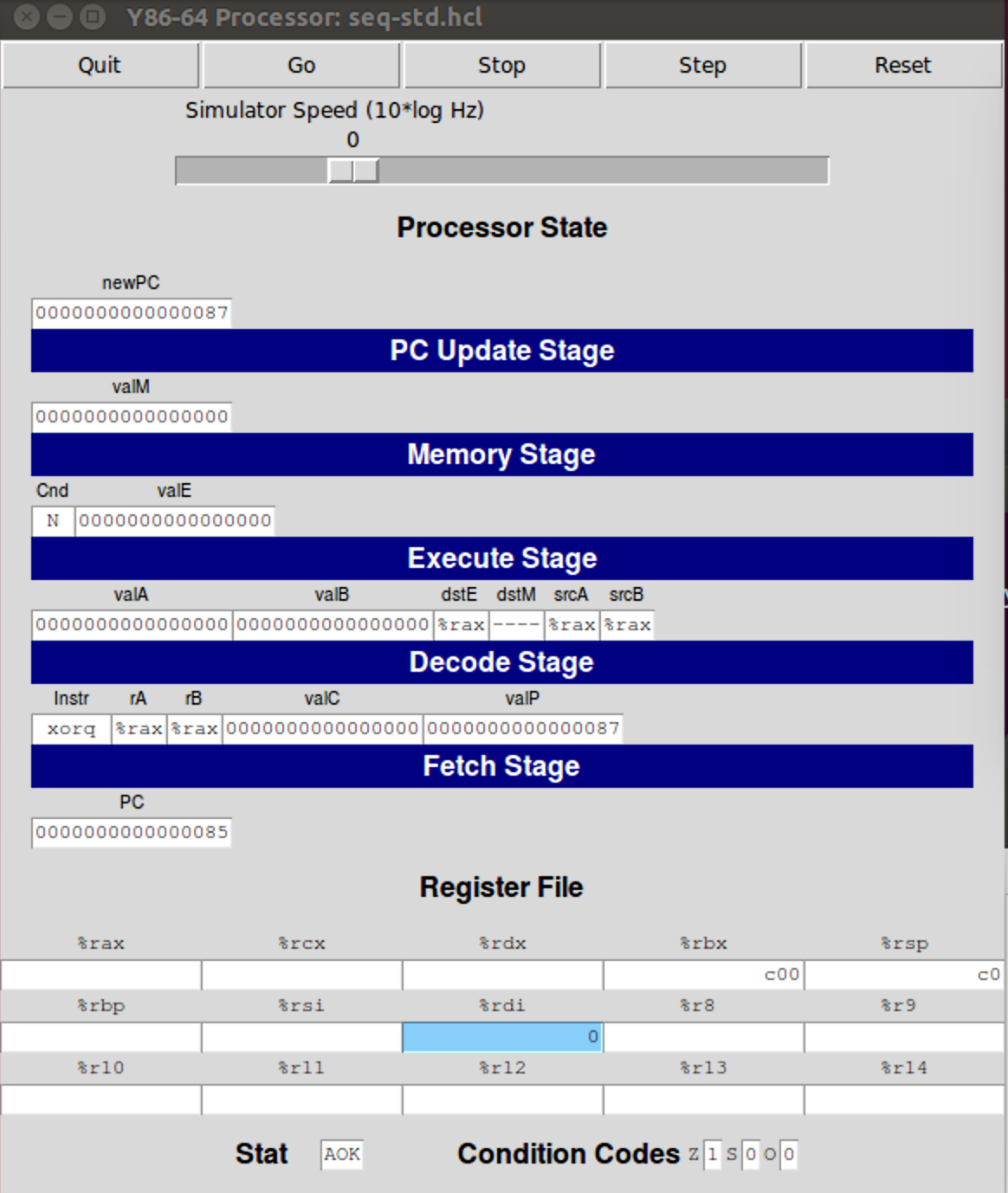
Like explained before, it increments the address %rdi, then put the deferenced value in %rdi. 30 is stored in 0x28, hence %rdi will be 30.

9. call rsum\_list.

The function is invoked again. This time the following pushq %rbx instruction will push the value in %rbx onto stack after decrementing the stack pointer (resulting in e0).

The function will be called recursively, until the base case, %rdi being zero, is reached. In this whole procedure, 0x00a, 0x0b0, 0xc00 are pushed onto stack respectively, and the stack pointer %rsp has value e8, e0, d8, c0 respectively (consider the first one an empty head node).

Then comes andq %rdi,%rdi, this time the zero flag is set and je will be executed.

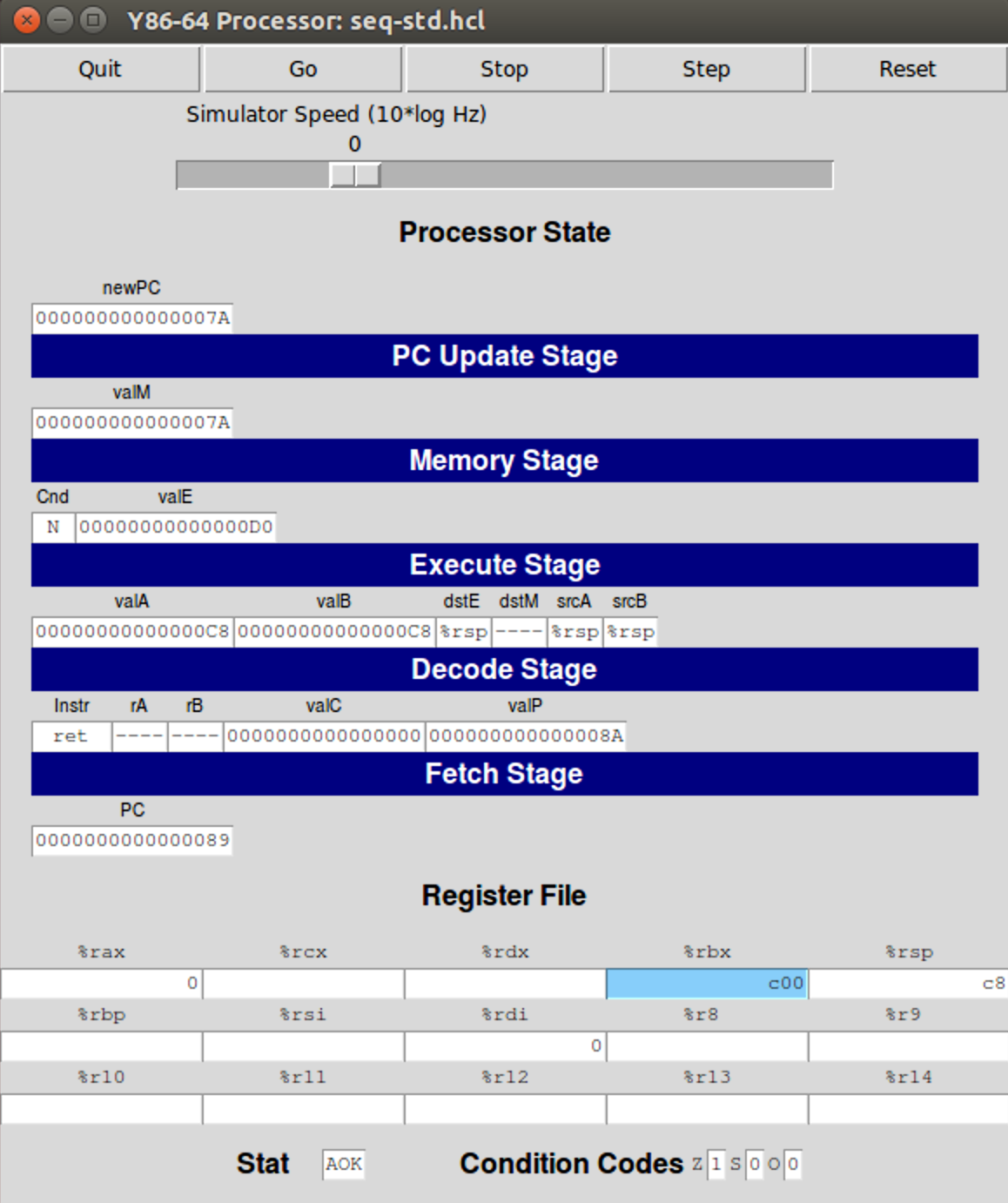


zreturn: xorq %rax,%rax

initialized the register %rax to zero.

done: popq %rbx

pops the stack and stores the stack top element in register %rbx, and increments the stack pointer %rsp by 8.



When a ret instruction is met, the instruction that was previously pushed onto stack will be assigned