## Lab 3: Simulating Y86-64 Program

## 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 Type

This is an individual project.

## 3 Instructions

NOTE: If using **ubuntu 12.10 32bit**:

(If using your own computer, make sure the the link in sources.list match your ubuntu version)

Please go to:

<https://www.cnblogs.com/wangzhigang/p/3893957.html>

* 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:

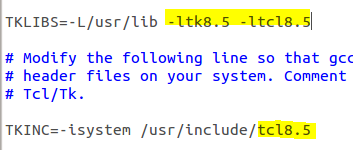
**# sudo apt-get update;**

**# sudo apt-get install flex bison;**

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

(In my experience, tcl8.6 and tk8.6 does not work)

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

****

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 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.

1. Change to the **sim** directory and build the Y86-64 tools:

**# make clean**

**# make**

1. 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 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 Check Succeeds.’ 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.

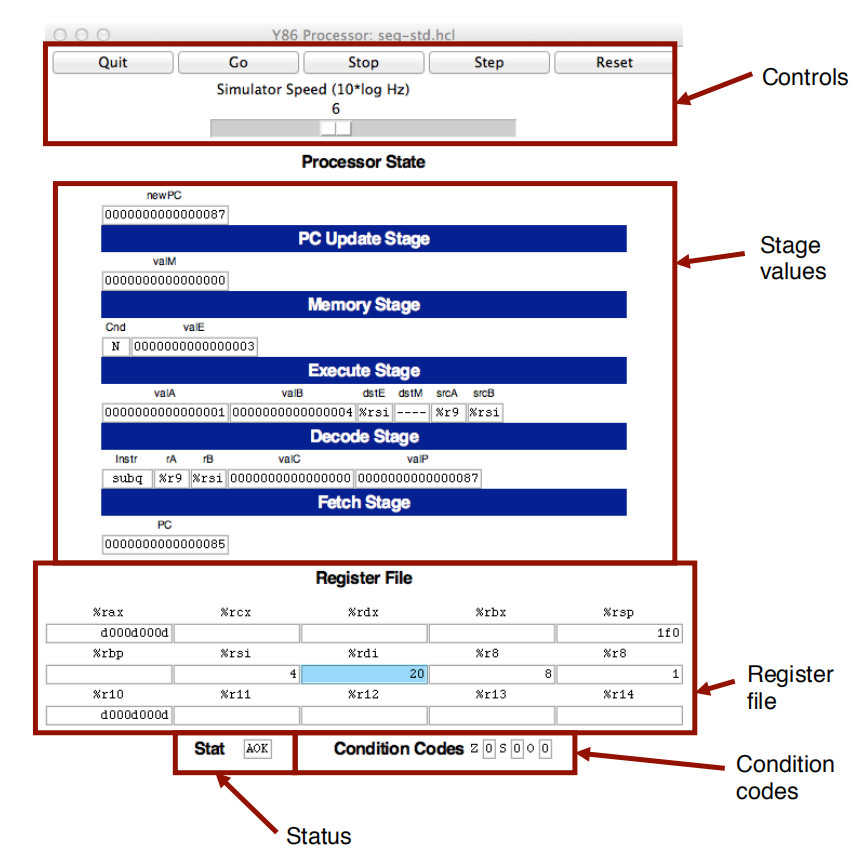


Figure 2 Main control panel for SEQ simulator

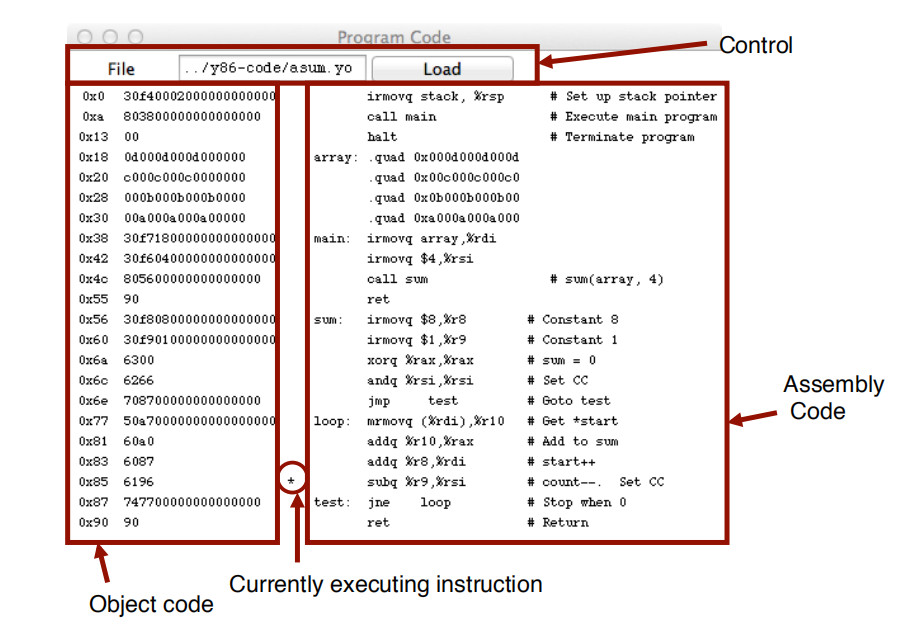


Figure 3 Code display window for SEQ simulator

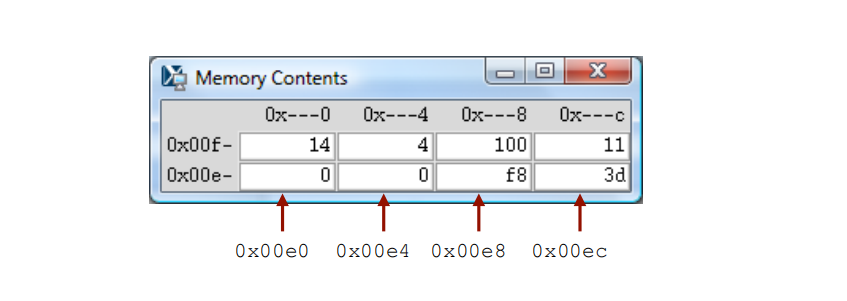


Figure 4 Memory display window for SEQ simulator

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 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 edit box 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 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*.

(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.

## 4 Handin

Results, Snapshots and Codes are inserted into Lab Report. You can hand in before the deadline.

This document describes the processor simulator that accompanies the presentation of the Y86-64 processor architecture in Chapter 4 of our textbook. It is edited by the TAs (Tan Xinhan976534762@qq.com, Tan Xinlin969551754@qq.com).