

# SE101 ICS Architecture Lab

## Optimizing the Performance of a Pipelined Processor

Assigned: Mar 25, Due: Apr 8, 11:59PM

TA Ruizhe Tong (tongruizhe@163.com) is responsible for this lab.

### 1 Introduction

In this lab, you will learn about the design and implementation of a pipelined Y86-64 processor, optimizing both it and a benchmark program to maximize performance. You are allowed to make any semantics preserving transformations to the benchmark program, or to make enhancements to the pipelined processor, or both. When you have completed the lab, you will have a keen appreciation for the interactions between code and hardware that affect the performance of your programs.

The lab is organized into three parts, each with its own handin. In Part A you will write some simple Y86-64 programs and become familiar with the Y86-64 tools. In Part B, you will extend the SEQ simulator with two new instructions. These two parts will prepare you for Part C, the heart of the lab, where you will optimize the Y86-64 benchmark program and the processor design.

### 2 Logistics

You will work on this lab alone.

Any clarifications and revisions to the assignment will be posted on the course Web page.

### 3 Handout Instructions

You can get the Architecture Lab with svn from the ICS Course Server (svn://ipads.se.sjtu.edu.cn/ics-se19/[account]). You can see a directory named “lab6” under your path. It contains several files: the `sim/` is your working directory in this lab and the `simguide.pdf` is a user guide to the Y86-64 Simulator provided by this lab.

You may realize that a lot more files and tools are provided this time compared to the previous labs. Fortunately, each directory contains a `README` file which describes the layout in detail. It is highly recommended to go through these files before you continue to the next part.

## 4 Part A

You will be working in directory `sim/misc` in this part.

Your task is to write three Y86-64 programs. The required behavior of these programs is defined by the example C functions in `examples.c`. Be sure to put your name and ID in a comment at the beginning of each program. You can test your programs by first assembling them with the program `YAS` and then running them with the instruction set simulator `YIS`. For example,

```
unix> ./yas sum.ys
unix> ./yis sum.yo
```

If you don't have `YAS` and `YIS` in the directory, first make them (You may need to install `flex` and `bison` to build these tools):

```
unix> make yas yis
```

In all of your Y86-64 functions, you should follow the x86-64 conventions for passing function arguments, using registers, and using the stack. This includes saving and restoring any callee-save registers that you use.

### **sum.ys: Iteratively sum linked list elements**

Write a Y86-64 program `sum.ys` that iteratively sums the elements of a linked list. Your program should consist 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. Test your program using the following three-element list:

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

### **rsum.ys: Recursively sum linked list elements**

Write a Y86-64 program `rsum.ys` that recursively sums the elements of a linked list. This code should be similar to the code in `sum.ys`, except that it should use a function `rsum_list` that recursively sums a list of numbers, as shown with the C function `rsum_list` in Figure 1. Test your program using the same three-element list you used for testing `sum.ys`.

```

1 /* linked list element */
2 typedef struct ELE {
3     int val;
4     struct ELE *next;
5 } *list_ptr;
6
7 /* sum_list - Sum the elements of a linked list */
8 int sum_list(list_ptr ls)
9 {
10     int val = 0;
11     while (ls) {
12         val += ls->val;
13         ls = ls->next;
14     }
15     return val;
16 }
17
18 /* rsum_list - Recursive version of sum_list */
19 int rsum_list(list_ptr ls)
20 {
21     if (!ls)
22         return 0;
23     else {
24         int val = ls->val;
25         int rest = rsum_list(ls->next);
26         return val + rest;
27     }
28 }
29
30 /* copy_block - Copy src to dest and return xor checksum of src */
31 int copy_block(int *src, int *dest, int len)
32 {
33     int result = 0;
34     while (len > 0) {
35         int val = *src++;
36         *dest++ = val;
37         result ^= val;
38         len--;
39     }
40     return result;
41 }

```

Figure 1: **C versions of the Y86-64 solution functions.** See `sim/misc/examples.c`

### **copy.y<sub>s</sub>: Copy a source block to a destination block**

Write a program (`copy.ys`) that copies a block of words from one part of memory to another (non-overlapping area) area of memory, computing the checksum (Xor) of all the words copied.

Your program should consist of code that sets up a stack frame, invokes a function `copy_block`, and then halts. The function should be functionally equivalent to the C function `copy_block` shown in Figure 1. Test your program using the following three-element source and destination blocks:

```
.align 8
# Source block
src:
    .quad 0x00a
    .quad 0x0b0
    .quad 0xc00

# Destination block
dest:
    .quad 0x111
    .quad 0x222
    .quad 0x333
```

## **5 Part B**

You will be working in directory `sim/seq` in this part.

Your task in Part B is to extend the SEQ processor to support the instruction `iaddq` (described in Homework problems 4.51 and 4.52). To add these instructions, you will modify the file `seq-full.hcl`, which implements the version of SEQ described in the CS:APP3e textbook. In addition, it contains declarations of some constants that you will need for your solution.

Your HCL file must begin with a header comment containing the following information:

- Your name and ID.
- A description of the computations required for the `iaddq` instruction. Use the descriptions of `irmovq` and `OPq` in Figure 4.18 in the CS:APP3e text as a guide.

### **Building and Testing Your Solution**

Once you have finished modifying the `seq-full.hcl` file, then you will need to build a new instance of the SEQ simulator (`ssim`) based on this HCL file, and then test it:

- *Building a new simulator:* You can use `make` to build a new SEQ simulator:

```
unix> make VERSION=full
```

This builds a version of `ssim` that uses the control logic you specified in `seq-full.hcl`. To save typing, you can assign `VERSION=full` in the Makefile.

- *Testing your solution on a simple Y86-64 program.* For your initial testing, we recommend running simple programs such as `asumi.yo` (testing `iaddq`) in TTY mode, comparing the results against the ISA simulation (If you have not built `asumi.yo`, go to build it first):

```
unix> ./ssim -t ../y86-code/asumi.yo
```

If the ISA test fails, then you may try to debug your implementation by single stepping the simulator in GUI mode (Refer to the section 10 on how to build a GUI mode simulator):

```
unix> ./ssim -g ../y86-code/asumi.yo
```

- *Retesting your solution using the benchmark programs.* Once your simulator is able to correctly execute small programs, then you can automatically test it on the Y86-64 benchmark programs in `../y86-code`:

```
unix> (cd ../y86-code; make testssim)
```

This will run `ssim` on the benchmark programs and check for correctness by comparing the resulting processor state with the state from a high-level ISA simulation. Note that *NONE* of these programs test the added instructions. You are simply making sure that your solution did not inject errors for the *original* instructions. See file `../y86-code/README` file for more details.

- *Performing regression tests.* Once you can execute the benchmark programs correctly, then you should run the extensive set of regression tests in `../ptest`. To test everything except `iaddq`:

```
unix> (cd ../ptest; make SIM=../seq/ssim)
```

To test your implementation of `iaddq`:

```
unix> (cd ../ptest; make SIM=../seq/ssim TFLAGS=-i)
```

For more information on the SEQ simulator refer to the handout *CS:APP3e Guide to Y86-64 Processor Simulators* (`simguide.pdf`).

## 6 Part C

You will be working in directory `sim/pipe` in this part.

The `ncopy` function in Figure 2 copies a `len`-element integer array `src` to a non-overlapping `dst`, returning a count of the number of positive integers contained in `src`. Figure 3 shows the baseline Y86-64 version of `ncopy`. The file `pipe-full.hcl` contains a copy of the HCL code for PIPE, along with a declaration of the constant value `IIADDQ`.

```

1 /*
2  * ncopy - copy src to dst, returning number of positive ints
3  * contained in src array.
4  */
5 int ncopy(int *src, int *dst, int len)
6 {
7     int count = 0;
8     int val;
9
10    while (len > 0) {
11        val = *src++;
12        *dst++ = val;
13        if (val > 0)
14            count++;
15        len--;
16    }
17    return count;
18 }

```

Figure 2: **C version of the ncopy function.** See `sim/pipe/ncopy.c`.

Your task in Part C is to modify `ncopy.y`s and `pipe-full.hcl` with the goal of making `ncopy.y`s run as fast as possible.

You will be handing in two files: `pipe-full.hcl` and `ncopy.y`s. Each file should begin with a header comment with the following information:

- Your name and ID.
- A high-level description of your code. In each case, describe how and why you modified your code.

## Coding Rules

You are free to make any modifications you wish, with the following constraints:

- Your `ncopy.y`s function must work for arbitrary array sizes. You might be tempted to hardwire your solution for 64-element arrays by simply coding 64 copy instructions, but this would be a bad idea because we will be grading your solution based on its performance on arbitrary arrays.
- Your `ncopy.y`s function must run correctly with YIS. By correctly, we mean that it must correctly copy the `src` block *and* return (in `%rax`) the correct number of positive integers.
- The assembled version of your `ncopy` file must not be more than 1000 bytes long. You can check the length of any program with the `ncopy` function embedded using the provided script `check-len.pl`:

```
unix> ./check-len.pl < ncopy.yo
```

```

1 #####
2 # ncopy.ys - Copy a src block of len words to dst.
3 # Return the number of positive words (>0) contained in src.
4 #
5 # Include your name and ID here.
6 #
7 # Describe how and why you modified the baseline code.
8 #
9 #####
10 # Do not modify this portion
11 # Function prologue.
12 # %rdi = src, %rsi = dst, %rdx = len
13 ncopy:
14
15 #####
16 # You can modify this portion
17 # Loop header
18 xorq %rax,%rax # count = 0;
19     andq %rdx,%rdx          # len <= 0?
20     jle Done                # if so, goto Done:
21
22 Loop: mrmovq (%rdi), %r10    # read val from src...
23     rmmovq %r10, (%rsi)     # ...and store it to dst
24     andq %r10, %r10        # val <= 0?
25     jle Npos               # if so, goto Npos:
26     irmovq $1, %r10
27     addq %r10, %rax         # count++
28 Npos: irmovq $1, %r10
29     subq %r10, %rdx         # len--
30     irmovq $8, %r10
31     addq %r10, %rdi         # src++
32     addq %r10, %rsi         # dst++
33     andq %rdx,%rdx         # len > 0?
34     jg Loop                # if so, goto Loop:
35 #####
36 # Do not modify the following section of code
37 # Function epilogue.
38 Done:
39     ret
40 #####
41 # Keep the following label at the end of your function
42 End:

```

Figure 3: **Baseline Y86-64 version of the ncopy function.** See `sim/pipe/ncopy.ys`.

- Your `pipe-full.hcl` implementation must be correct. It should pass the regression tests in `../y86-code` and `../ptest` (see below).

Other than that, you are free to implement the `iaddq` instruction if you think that will help. You may make any semantics preserving transformations to the `ncopy.y8` function, such as reordering instructions, replacing groups of instructions with single instructions, deleting some instructions, and adding other instructions. You may find it useful to read about loop unrolling in Section 5.8 of CS:APP3e.

## Building and Running Your Solution

In order to test your solution, you will need to build a driver program that calls your `ncopy` function. We have provided you with the `gen-driver.pl` program that generates a driver program for arbitrary sized input arrays. For example, typing

```
unix> make drivers
```

will construct the following two useful driver programs:

- `sdriver.y8`: A *small driver program* that tests an `ncopy` function on small arrays with 4 elements. If your solution is correct, then this program will halt with a value of 2 in register `%rax` after copying the `src` array.
- `ldriver.y8`: A *large driver program* that tests an `ncopy` function on larger arrays with 63 elements. If your solution is correct, then this program will halt with a value of 31 (0x1f) in register `%rax` after copying the `src` array.

Each time you modify your `ncopy.y8` program, you can rebuild the driver programs by typing

```
unix> make drivers
```

Each time you modify your `pipe-full.hcl` file, you can rebuild the simulator by typing

```
unix> make psim VERSION=full
```

To test your solution in GUI mode on a small 4-element array, type

```
unix> ./psim -g sdriver.y8
```

To test your solution on a larger 63-element array, type

```
unix> ./psim -g ldriver.y8
```

Once your simulator correctly runs your version of `ncopy.y8` on these two block lengths, you will want to perform the following additional tests:



- *Testing your driver files on the ISA simulator.* Make sure that your `ncopy.y`s function works properly with YIS:

```
unix> make drivers
unix> ../misc/yis sdriver.yo
```

- *Testing your code on a range of block lengths with the ISA simulator.* The Perl script `correctness.pl` generates driver files with block lengths from 0 up to some limit (default 65), plus some larger sizes. It simulates them (by default with YIS), and checks the results. It generates a report showing the status for each block length:

```
unix> ./correctness.pl
```

This script generates test programs where the result count varies randomly from one run to another, and so it provides a more stringent test than the standard drivers.

If you get incorrect results for some length  $K$ , you can generate a driver file for that length that includes checking code, and where the result varies randomly:

```
unix> ./gen-driver.pl -f ncopy.y -n K -rc > driver.y
unix> make driver.yo
unix> ../misc/yis driver.yo
```

The program will end with register `%rax` having the following value:

**0xaaaa** : All tests pass.

**0xbbbb** : Incorrect count

**0xcccc** : Function `ncopy` is more than 1000 bytes long.

**0xdddd** : Some of the source data was not copied to its destination.

**0xeeee** : Some word just before or just after the destination region was corrupted.

- *Testing your pipeline simulator on the benchmark programs.* Once your simulator is able to correctly execute `sdriver.y`s and `ldriver.y`s, you should test it against the Y86-64 benchmark programs in `../y86-code`:

```
unix> (cd ../y86-code; make testpsim)
```

This will run `psim` on the benchmark programs and compare results with YIS.

- *Testing your pipeline simulator with extensive regression tests.* Once you can execute the benchmark programs correctly, then you should check it with the regression tests in `../ptest`. For example, if your solution implements the `iaddq` instruction (not compulsory), then

```
unix> (cd ../ptest; make SIM=../pipe/psim TFLAGS=-i)
```

- *Testing your code on a range of block lengths with the pipeline simulator.* Finally, you can run the same code tests on the pipeline simulator that you did earlier with the ISA simulator

```
unix> ./correctness.pl -p
```

## 7 Evaluation

The lab is worth 190 points: 30 points for Part A, 60 points for Part B, and 100 points for Part C.

### Part A

Part A is worth 30 points, 10 points for each Y86-64 solution program. Each solution program will be evaluated for correctness, including proper handling of the stack and registers, as well as functional equivalence with the example C functions in `examples.c`.

The programs `sum.y8` and `rsum.y8` will be considered correct if the graders do not spot any errors in them, and their respective `sum_list` and `rsum_list` functions return the sum `0xcba` in register `%rax`.

The program `copy.y8` will be considered correct if the graders do not spot any errors in them, and the `copy_block` function returns the checksum `0xcba` in register `%rax`, copies the three 64-bit values `0x00a`, `0x0b0`, and `0xc00` to the 24 bytes beginning at address `dest`, and does not corrupt other memory locations.

### Part B

This part of the lab is worth 60 points:

- 10 points for your description of the computations required for the `iaddq` instruction.
- 20 points for passing the benchmark regression tests in `y86-code`, to verify that your simulator still correctly executes the benchmark suite.
- 30 points for passing the regression tests in `pctest` for `iaddq`.

### Part C

This part of the Lab is worth 100 points: **You will not receive any credit if either your code for `ncopy.y8` or your modified simulator fails any of the tests described earlier.**

- 20 points each for your descriptions in the headers of `ncopy.y8` and `pipe-full.hcl` and the quality of these implementations.
- 60 points for performance. To receive credit here, your solution must be **CORRECT**, as defined earlier. That is, `ncopy` runs correctly with your pipeline implementation as well as YIS, the assembled version of your `ncopy` file must not be more than 1000 bytes long, and `pipe-full.hcl` passes all tests in `y86-code` and `pctest`.

We will express the performance of your function in units of *cycles per element* (CPE). That is, if the simulated code requires  $C$  cycles to copy a block of  $N$  elements, then the CPE is  $C/N$ . The PIPE simulator displays the total number of cycles required to complete the program. The baseline version

of the `ncopy` function running on the standard PIPE simulator with a large 63-element array requires 897 cycles to copy 63 elements, for a CPE of  $897/63 = 14.24$ .

Since some cycles are used to set up the call to `ncopy` and to set up the loop within `ncopy`, you will find that you will get different values of the CPE for different block lengths (generally the CPE will drop as  $N$  increases). We will therefore evaluate the performance of your function by computing the average of the CPEs for blocks ranging from 1 to 64 elements. You can use the Perl script `benchmark.pl` in the `pipe` directory to run simulations of your `ncopy.hs` code over a range of block lengths and compute the average CPE. Simply run the command

```
unix> ./benchmark.pl
```

to see what happens. For example, the baseline version of the `ncopy` function has CPE values ranging between 29.00 and 14.27, with an average of 15.18. Note that this Perl script does not check for the correctness of the answer. Use the script `correctness.pl` for this.

You should be able to achieve an average CPE of less than 9.00. Our best version averages 7.48. If your average CPE is  $c$ , then your score  $S$  for this portion of the lab will be:

$$S = \begin{cases} 0, & c > 10.5 \\ 20 \cdot (10.5 - c), & 7.50 \leq c \leq 10.50 \\ 60, & c < 7.50 \end{cases}$$

By default, `benchmark.pl` and `correctness.pl` compile and test `ncopy.hs`. Use the `-f` argument to specify a different file name. The `-h` flag gives a complete list of the command line arguments.

## 8 Part-C Evaluation

The final score of Part-C is based on the TA's computer. If you want to know your final score, you can commit `ncopy.hs` and `pipe-full.hcl` to `svn`. The score will be published on:

<http://ipads.se.sjtu.edu.cn/courses/ics/labs/archlab/arch-score.html>

**Note: the web page will not be updated everyday.**

## 9 Handin Instructions

- You will be handing in three sets of files:
  - Part A: `sum.hs`, `rsum.hs`, and `copy.hs` (Don't forget to use `svn add` to put them under version control).
  - Part B: `seq-full.hcl`.
  - Part C: `ncopy.hs` and `pipe-full.hcl`.
- Make sure you have included your name and ID in a comment at the top of each of your handin files.

## 10 Hints

- The `psim` and `ssim` simulators terminate with a segmentation fault if you ask them to execute a file that is not a valid Y86-64 object file.
- You may need to install `flex` and `bison`.

```
sudo apt-get install flex bison
```

### Install Tcl/Tk

Running the simulator in GUI mode gives a visualization of how the program is executed at runtime. Not only does it help to debug your implementation in this lab but enhance your understanding of what we have learned in this chapter as well. To build a GUI mode simulator, you can refer to the `Makefile` that build the simulator for details. You should have Tcl/Tk 8.5 installed on your system as well as its headers and libraries.

Run the following command to install Tcl/Tk 8.5 as well as its headers and libraries,

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

Debian 10 has removed `tcl8.5-dev` and `tk8.5-dev` from apt source, so the command above will fail. **If you use Debian 10**, you need to perform the following commands as **root**.

```
apt_source="deb http://mirrors.ustc.edu.cn/debian stretch main contrib"
echo $apt_source > /etc/apt/sources.list.d/stretch.list
echo "APT::Default-Release \"buster\";" > /etc/apt/apt.conf.d/99stretch
apt update
apt -qqy install libx11-dev libxext-dev libxft-dev libxss-dev dpkg-dev
apt -qqy -t stretch install tcl8.5-dev tk8.5-dev
```

After installing Tcl/Tk, you need to uncomment GUI-mode-related lines in the `Makefile` to build GUI-mode simulator.

```
GUIMODE=-DHAS_GUI
TKLIBS=-L/usr/lib -ltk8.5 -ltcl8.5
TKINC=-isystem /usr/include/tcl8.5
```

If you have any question, please send an email to Ruizhe Tong (tongruizhe@163.com).