

18-600, Fall 2017  
Architecture Lab  
Assigned: Thursday, Sep. 28, 11:59PM PDT  
Due: Thursday, Oct. 12, 11:59PM PDT  
Last Day to Hand in: Sunday, Oct. 15, 11:59PM PDT

## 1 Introduction

In this lab assignment, you will learn about performing optimizations both below and above the Instruction Set Architecture (ISA), and exploring the design space of superscalar out-of-order (O3) processors. You will be modifying both high level code of several benchmarks and parameters of an superscalar O3 CPU model using a simulation tool to identify bottlenecks, and improve performance. When you have completed the lab assignment, you will gain keen appreciation for the interactions between object code and execution hardware that affect the performance of your programs.

The lab is organized into three parts. In Part A you will write some simple Y86-64 programs and become familiar with the Y86-64 tools. In Part B, you will learn to optimize the Y86-64 benchmark program. In Part C, you will optimize two specific parameters of a superscalar O3 CPU model in order to improve the performance and performance/cost (efficiency) for running two specific benchmarks.

## 2 Logistics

You will work on this lab alone.

Any clarifications and revisions to the assignment will be posted on Piazza.

## 3 Handout Instructions

1. Start by downloading the handout `arclab-handout.tar` from autolab to a (protected) directory in which you plan to do your work.
2. Then give the command: `tar xvf arclab-handout.tar`. This will cause the following files to be unpacked into the directory: `README`, `mm.c`, `daxpy.c`, `sim.tar`, `Makefile`, `simguide.pdf`, `o3_mm.py`, `o3_daxpy.py`, `o3_star.py`, `arclab.pdf`, and `driver.py`.

3. Next, give the command `tar xvf sim.tar`. This will create the directory `sim`, which contains your personal copy of the Y86-64 tools. You will be doing all of your work inside this directory.
4. Finally, change to the `sim` directory and build the Y86-64 tools:

```
unix> cd sim
unix> make clean; make
```

## 4 Part A

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

Your task is to write and simulate the following 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`.

Example:

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

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.

We recommend checking out the example programs in the `y86-code` folder for inspiration.

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

Write a Y86-64 program `sum.y` 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
```

```

1 /* linked list element */
2 typedef struct ELE {
3     long val;
4     struct ELE *next;
5 } *list_ptr;
6
7 /* sum_list - Sum the elements of a linked list */
8 long sum_list(list_ptr ls)
9 {
10     long 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 long rsum_list(list_ptr ls)
20 {
21     if (!ls)
22         return 0;
23     else {
24         long val = ls->val;
25         long 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 long copy_block(long *src, long *dest, long len)
32 {
33     long result = 0;
34     while (len > 0) {
35         long 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`

### **rsum.y<sub>s</sub>: 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 `list.ys`.

### **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 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
```

## **Evaluation**

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.ys` and `rsum.ys` 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.ys` will be considered correct if the graders do not spot any errors in them, and the `copy_block` function returns the sum `0xcba` in register `%rax`, copies the three 64-bit values `0x00a`, `0x0b`, and `0xc` to the 24 bytes beginning at address `dest`, and does not corrupt other memory locations.

## **5 Part B**

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

```

1 /*
2  * ncopy - copy src to dst, returning number of positive ints
3  * contained in src array.
4  */
5 word_t ncopy(word_t *src, word_t *dst, word_t len)
6 {
7     word_t count = 0;
8     word_t 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`.

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`.

Your task in Part B is to modify `ncopy.y`s with the goal of making `ncopy.y`s run as fast as possible.

You will be handing in `ncopy.y`s with a header comment with the following information:

- Your name and ID.
- A high-level description of your code, and 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       # count++
27         addq %r10, %rax
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`.

You may make any semantics preserving transformations to the `ncopy.y`s function, such as reordering instructions, replacing groups of instructions with single instructions, deleting some instructions, and adding other instructions.

## Hints for Part B

- You may find it useful to read about loop unrolling in Section 5.8 of CS:APP3e.
- You may find it useful to read about jump tables or other methods of control in Section 3.6
- The majority of your optimizations will occur in `ncopy.y`s, so we recommend focusing your attention there

## 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.yo`: 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.yo`: 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.y`s program, you can rebuild the driver programs by typing

```
unix> make drivers
```

You can build the default pipeline simulator for this lab by typing

```
unix> make psim
```

If you want to rebuild the simulator and the driver programs, type

```
unix> make
```

For your initial testing, we recommend running your solution in TTY mode, comparing the results against the ISA simulation:

```
unix> ./psim -t sdriver.yo
```

If the ISA test fails, then you should debug your implementation by single stepping the simulator in GUI mode (more info on using GUI mode in hints at end of handout). To test your solution in GUI mode on a small 4-element array, type

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

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

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

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

- *Testing your driver files on the ISA simulator.* Make sure that your `ncopy.yo` 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.yo -n K -rc > driver.yo
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 program against the default pipeline.* To ensure that your program is using instructions that are supported by the default pipeline model, execute

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

This performs the same set of tests as running `./correctness.pl` but using the default pipeline model.



## Evaluation

This part of the Lab is worth 50 points: **You will not receive any credit if your code for `ncopy.y` fails any of the tests described earlier.**

- 5 points for your description in the header of `ncopy.y` and the quality of the description.
- 45 points for performance. To receive credit here, your solution must be correct, as defined earlier. That is, `ncopy` runs correctly with YIS.

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.y` 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 10. Our (TAs) best version averages 6.26. If your average CPE is  $c$ , then your score  $S$  for this portion of the lab will be:

$$S = \begin{cases} 0, & c \geq 12.5 \\ 11.25 \cdot (12.5 - c), & 8.5 \leq c < 12.5 \\ 45, & c \leq 8.5 \end{cases}$$

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

## 6 Gem5

For Part C, we will be using a simulator Gem5<sup>1</sup> with x86 CPU model to collect performance metrics for two benchmark programs to analyze their performance bottlenecks. Gem5 is widely used by researchers and developers in academia and industry, supports the standard PC platform, and provides plenty of useful

---

<sup>1</sup>[http://www.gem5.org/Main\\_Page](http://www.gem5.org/Main_Page)

metrics for real workloads. You will run the binaries of the benchmarks using the Gem5 simulator with several CPU model configurations. A Gem5 tutorial will be conducted in recitation.

Among the many metrics provided by Gem5, we will focus on Cycles Per Instruction (CPI) as the performance measure, and a *hardware cost* ( $C$ ) to evaluate the efficiency of the machine model. We define *efficiency* ( $E$ ) as  $E = CPI \times C^{1/3}$ . For this lab we will calculate  $C$  based on the sizes of two specific buffers (ROB and IQ) used in Out-of-Order (O3) pipeline.

## 6.1 Superscalar O3 Processor Pipeline

In this part you will be working with different parameters of a superscalar out-of-order (O3) pipeline to optimize performance for two benchmarks.

For each of these benchmarks, we have provided configuration files mentioned in Table 1 which you can edit, and run with Gem5 to see performance changes. Below is a list of parameters which can be tweaked to improve performance using Gem5.

1. numROBEntries: Number of reorder buffer entries.
2. numIQEntries: Number of instruction queue entries (reservation station entries).

To check the CPI using the default set of parameters, run the `driver.py`:

```
unix> python driver.py -g
```

- All parameters and their values for the simulation run can be found in a configuration file `config.ini` created in the output folder. For example the output folder for naive matrix multiplication is:  
`${ARCLAB-ROOT}/out/o3_mm_naive`.
- `${ARCLAB-ROOT}/out/o3-{benchmark}` also contains statistics for each run in a file named `stats.txt`. This file should help you analyze how changing each parameter improves or degrades performance. Since you have to provide reasoning for why changing one or multiple parameters improves performance, please monitor `stats.txt` while modifying the CPU model parameters.

Each Gem5 configuration file listed in Table 1 has a block of code after a comment `# Parameter values` (as shown on line 11) to make your changes:

```
1 def create_cpu(options, cpu_id):
2     # DerivO3CPU is the configurable out-of-order
3     # CPU model supplied by gem5
4     the_cpu = DerivO3CPU(cpu_id=cpu_id)
5     icache = L1_ICache(size=options.ll1_size,
6                        assoc=options.ll1_assoc)
7     dcache = L1_DCache(size=options.ll2_size,
8                        assoc=options.ll2_assoc)
9     the_cpu.branchPred = LocalBP()
10
```

<i>Benchmark</i>	<i>Gem5 Config. file</i>
Matrix Multiplication	<code>o3_mm.py</code>
DAXPY	<code>o3_daxpy.py</code>

Table 1: Gem5 configuration files for the two benchmarks

<i>CPU Model Name</i>	<i>{numROBEntries, numIQEntries}</i>
1. Small	{8, 4}
2. Medium	{16, 8}
3. Large	{32, 16}
4. XLarge	{64, 32}

Table 2: CPU Models

```

11     # Parameter values
12     the_cpu.numROBEntries = 4
13     the_cpu.numIQEntries = 2

```

For example, to modify `numROBEntries` from 4 to 6, you need to edit line 12:

```

11     # Parameter values
12     the_cpu.numROBEntries = 6
13     the_cpu.numIQEntries = 2

```

After saving your changes, you can run:

```
unix> make rmgem && python driver.py -g
```

to get a new set of results. Please make sure to reference every parameter with the object `the_cpu` which represents the superscalar out-of-order CPU model provided by Gem5.

## 7 Part C

In this part you will run two benchmarks using Gem5. To test, and get the CPI for all benchmarks run `python driver.py -g`. For any subsequent runs using Gem5 it is important to delete the Gem5 results from the previous run using:

```
unix> make rmgem
```

If you want to retain previous results, make a copy of the folder `${ARCLAB-ROOT}/out/o3-{benchmark}` somewhere in your home directory on AFS.

### 7.1 Benchmark 1: Matrix Multiplication

Sweep across the four CPU models by modifying parameters: `numROBEntries`, `numIQEntries` in `o3_mm.py`. The exact values for each model are given in Table 2. For each model run the driver and note

the CPI, cost ( $C$ ), and efficiency ( $E$ ) calculated using:

$$E = CPI \times C^{1/3} \quad (1)$$

where cost  $C$  is sum of the two parameter values, i.e. the total number of ROB and IQ entries. The *Small CPU Model* parameter values are already set in the handout. Report the CPI,  $C$ , and  $E$  for the four CPU models, by setting `numROBEntries`, and `numIQEntries` in `o3_mm.py` according to the values specified in Table 2. Identify the best CPU model in terms of having the lowest value for  $E$ . Do provide a comment in your python script which states the name of the best CPU model for matrix multiplication along with its CPI,  $C$ , and  $E$  collected from `driver.py`. You will need to generate the binary by issuing the following command in the root directory of the handout:

```
unix> make mm
```

## 7.2 Benchmark 2: DAXPY

Sweep across the four CPU models, as before, but make modifications in `o3_daxpy.py`. Report the results from the four CPU models by setting the two parameter values according to the values in Table 2. Provide a comment stating the name of the best CPU model in `o3_daxpy.py` along with metrics collected from `driver.py` for all four machine models. To generate the binary run `make daxpy`.

## 7.3 Star Configuration

Start from `numROBEntries=32`, and `numROBEntries=16` as provided in `o3_star.py`. Now assume you have an extra quota of 8 entries that you can use to increase the sizes of the ROB and/or IQ. This means that starting with  $32 + 16$  entries you can now increase the total cost  $C$  from 48 to up to 56 (this total can be split between ROB and IQ per your choice).

Your task is to find the best configuration which works well for both the benchmarks in terms of efficiency  $E$ . Find the best balance between the two parameters, and report the best observed set of values in `o3_star.py`. The best configuration for each benchmark may not require using all 8 additional entries. For this part it will be helpful to look at `ROBFullEvents` and `IQFullEvents` in `stats.txt` for each of the benchmarks. Run `driver` with `-gs` to ensure that both benchmarks run using `o3_star.py`:

```
unix> python driver.py -gs
```

## Evaluation

Part C is worth 30 points, 10 points for reporting the performance results for the two benchmarks on the four machine models, and identifying the best model for each benchmark; 10 points for reporting the best star configuration for each of the two benchmarks; and 10 points for analyzing results obtained while coming up with the best star configurations. Your analysis should contain empirical results like observed CPI values across different configurations, efficiency  $E$  values etc., and clear reasoning as to why a particular parameter or both are more relevant to reduce CPI in finding the star configuration.

You are welcome to refer and cite material from any of the course textbooks or lecture notes for the description.

## 8 The Driver

Finally, when you are ready to test your whole lab, navigate to your `arclab-handout` directory and run:

```
unix> python driver.py
```

This will run the same driver that autolab uses to grade your solution, and assign a score for both parts A and B. Part C will be graded manually by TAs.

## 9 Handin Instructions

- You will be handing in two sets of files:
  - Part A: `sum.py`, `rsum.py`, and `copy.py`.
  - Part B: `ncopy.py`.
  - Part C: `o3_mm.py`, `o3_daxpy.py`, `o3_star.py`.
- Make sure you have included your name and ID in a comment at the top of each of your handin files.
- Each file should also have a high-level description of your code. In each case, describe how and why you modified your code.
- Detailed comments are to be provided for parameters modified as the case with `o3_star.py`. Comments for `o3_mm.py` and `o3_daxpy.py` are optional. Although do report the metrics observed using `driver.py` for the different CPU Models.
- To create a handin of your files for autolab, go to your `arclab-handout` directory and type:

```
unix> make handin
```

This simply creates a `arclab-handin.tar` file containing the specified files, which then will be submitted to autolab.

## 10 Hints

- By design, both `sdriver.py` and `ldriver.py` are small enough to debug with in GUI mode. Some find it easiest to debug in GUI mode, and we suggest that you use it if you get stuck. GUI mode requires that secure X-forwarding be enabled for your `ssh` session with the shark machines. On unix machines, this generally means using `ssh -Y` instead of just `ssh`. Since this is largely dependent on your own computing environment, please be advised that the TAs may be no more competent than you at figuring out how to enable it on your machine. Below though are a collection of tips for getting it to work on various computing environments.
- If you have a Unix environment, you should be able to just use `ssh -Y` to connect

- If you have a Mac running a version of OS X, we recommend using XQUARTZ, as X11 is no longer included with OS X but might be if you have an older mac. Then, using the XQUARTZ terminal use `ssh -Y` to connect
- If you have a Windows machine, you have many random options. We have had success with Putty and Xming for Windows 10. A piazza post will be made for more info on this method.
- If you running in GUI mode on a Unix server, you may want to make sure that you have initialized the `DISPLAY` environment variable:

```
unix> setenv DISPLAY myhost.edu:0
```

- With some X servers, the “Program Code” window begins life as a closed icon when you run `psim` or `ssim` in GUI mode. Simply click on the icon to expand the window.
- With some Microsoft Windows-based X servers, the “Memory Contents” window will not automatically resize itself. You’ll need to resize the window by hand.
- 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.
- CPI may vary slightly across the shark machines. Although it shouldn’t change the best CPU model for each benchmark.
- For finding the star configuration, stick with even values for each of the two parameters.