# 4190.308: Computer Architecture, Fall 2016
# Lab 3: SEQ and PIPE implementations

November 11, 2016

Due: November 25, 09:59AM

## 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 transformation 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 two parts, each with its own handin. In Part A, you will extend the sequential simulator with a new instruction. This part will prepare you for Part B, the heart of the lab, where you will optimize the Y86-64 benchmark program and the processor design.

## 2 Handout Instructions

1. Start by downloading the file `Lab3.tar` from eTL.
2. Then give the command: `tar xvf Lab3.tar`. This will cause the following files to be unpacked into the directory: `sim.tar`, `archlab.pdf`, and `simguide.pdf`.
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
```

## 3 Part A: Sequential Implementation

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

Your task in Part A is to extend the SEQ processor to support the `iaddq`, described in Homework problems 4.51 and 4.52. To add this 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.

### 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`), comparing the results against the ISA simulation:

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

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

## 4 Part B: Pipelined Implementation

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

The `ncopy` function in Figure 1 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 2 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`.

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

You will be handing in two files: `pipe-full.hcl` and `ncopy.ys`.

### Coding Rules

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

```
/*
 * ncopy - copy src to dst, returning number of positive ints
 * contained in src array.
 */
word_t ncopy(word_t *src, word_t *dst, word_t len)
{
    word_t count = 0;
    word_t val;

    while (len > 0) {
        val = *src++;
        *dst++ = val;
        if (val > 0)
            count++;
        len--;
    }
    return count;
}
```

Figure 1: **C version of the `ncopy` function.** See `sim/pipe/ncopy.c`.

- Your `ncopy.ys` 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.ys` 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
  ```

- Your `pipe-full.hcl` implementation must pass the regression tests in `../y86-code` and `../ptest` (without the `-i` flag that tests `iaddq`).

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.ys` 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

```
##################################################################
# ncopy.ys - Copy a src block of len words to dst.
# Return the number of positive words (>0) contained in src.
#
##################################################################
# Do not modify this portion
# Function prologue.
# %rdi = src, %rsi = dst, %rdx = len
ncopy:

##################################################################
# You can modify this portion
        # Loop header
        xorq %rax,%rax          # count = 0;
        andq %rdx,%rdx          # len <= 0?
        jle Done                # if so, goto Done:

Loop:   mrmovq (%rdi), %r10     # read val from src...
        rmmovq %r10, (%rsi)     # ...and store it to dst
        andq %r10, %r10         # val <= 0?
        jle Npos                # if so, goto Npos:
        irmovq $1, %r10
        addq %r10, %rax         # count++
Npos:   irmovq $1, %r10
        subq %r10, %rdx         # len--
        irmovq $8, %r10
        addq %r10, %rdi         # src++
        addq %r10, %rsi         # dst++
        andq %rdx,%rdx          # len > 0?
        jg Loop                 # if so, goto Loop:
##################################################################
# Do not modify the following section of code
# Function epilogue.
Done:
        ret
##################################################################
# Keep the following label at the end of your function
End:
```

Figure 2: **Baseline Y86-64 version of the `ncopy` function.** See `sim/pipe/ncopy.ys`.

```
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.ys` 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
```

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

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

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

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

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

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

## 5 Evaluation

The lab is worth 100 points: 40 points for Part A and 60 points for Part B.

### Part A

This part of the lab is worth 40 points:

- 40 points for passing the simple Y86-64 program test (and other tests not released to you), to verify that your simulator correctly executes the Y86-64 instruction.

### Part B

This part of the Lab is worth 60 points: **You will not receive any credit if either your code for `ncopy.ys` or your modified simulator fails the tests described earlier.**

- 60 points for performance. To receive credit here, your solution must be correct, as defined earlier.

  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.ys` 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.ys`. Use the `-f` argument to specify a different file name. The `-h` flag gives a complete list of the command line arguments.

## 6 Handin Instructions

- You will be handing in two sets of files:
  - Part A: `seq-full.hcl`.
  - Part B: `ncopy.ys` and `pipe-full.hcl`.
- Archive your files into *YourStudentID*.tar by the following command

```
unix>  tar cvf YourStudentID.tar sim/seq/seq-full.hcl \
                 sim/pipe/pipe-full.hcl sim/pipe/ncopy.ys
```

- Send e-mail to `snu.comarch.2016@gmail.com` with attaching the archive.
- Subject of e-mail: [CA Lab3] *YourStudentID*
- Note that the due date is 09:59AM on November 25, 2016