CS 261, Fall 2017
Optimizing the Performance of a Pipelined Processor
Assigned: Oct. 9, 2017
Part A (Homework #5) Due: Oct. 16 2017 @ 5:00 pm
Part B (Homework #6) Due: Oct. 23, 2017 @ 5:00 pm
Part C (Project #2) Due: Nov. 1, 2017 @ 11:59 pm

1 Introduction

In this project, 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 project, you will have a keen appreciation for the interactions between code and hardware that affect the performance of your programs.

The project 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 a new instruction. These two parts will prepare you for Part C, the heart of the project, where you will optimize the Y86-64 benchmark program and the processor design.

2 Logistics

You will work on this project alone.

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

Before you submit your work, you should read Section 7 $Evaluation$ and Section 8 $SubmissionRules$.

3 Handout Instructions

All documents that are necessary for this project are on Piazza. You will also see a link to download a tar file, `archlab-handout.tar`.

1. Start by copying the file `archlab-handout.tar` to a (protected) directory in which you plan to do your work.

2. Then give the command: `tar xvf archlab-handout.tar`. This `tar` command will extract the contents of the archive into your current working directory. There should now be a folder in your current working directory called `archlab-handout` and it will contain the following files: `README`, `Makefile`, `sim.tar`, `archlab.pdf`, and `simguide.pdf`.

3. Next, change directories into `archlab-handout` and 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 UIC netID in a comment at the beginning of each program. You can test your programs by first assemblying them with the program YAS and then running them with the instruction set simulator 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 0x00d
        .quad ele2
ele2:
        .quad 0x0e0
        .quad ele3
ele3:
```

```
        .quad 0xf00
        .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`.

`copy.ys`: 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 Figure 1. Test your program using the following three-element source and destination blocks:

```
.align 8
# Source block
src:
        .quad 0x00d
        .quad 0x0e0
        .quad 0xf00

# 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 `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.

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

- Your name and UIC netID.

```c
/* linked list element */
typedef struct ELE {
    int val;
    struct ELE *next;
} *list_ptr;

/* sum_list - Sum the elements of a linked list */
int sum_list(list_ptr ls)
{
    int val = 0;
    while (ls) {
        val += ls->val;
        ls = ls->next;
    }
    return val;
}

/* rsum_list - Recursive version of sum_list */
int rsum_list(list_ptr ls)
{
    if (!ls)
        return 0;
    else {
        int val = ls->val;
        int rest = rsum_list(ls->next);
        return val + rest;
    }
}

/* copy_block - Copy src to dest and return xor checksum of src */
int copy_block(int *src, int *dest, int len)
{
    int result = 0;
    while (len > 0) {
        int val = *src++;
        *dest++ = val;
        result ^= val;
        len--;
    }
    return result;
}
```

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

- 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:

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

  If the ISA test fails, then you should debug your implementation by single stepping the simulator in GUI mode:

  ```
  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` and `leave`:

  ```
  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`).

```
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`.

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`.

Your task in Part C 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`. Each file should begin with a header comment with the following information:

- Your name and UIC netID.
- 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.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.

```
1  ##################################################################
2  # ncopy.ys - Copy a src block of len ints to dst.
3  # Return the number of positive ints (>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 ncopy:  pushl %ebp              # Save old frame pointer
13         rrmovl %esp,%ebp        # Set up new frame pointer
14         pushl %esi              # Save callee-save regs
15         pushl %ebx
16         pushl %edi
17         mrmovl 8(%ebp),%ebx     # src
18         mrmovl 16(%ebp),%edx    # len
19         mrmovl 12(%ebp),%ecx    # dst
20
21 ##################################################################
22 # You can modify this portion
23         # Loop header
24         xorl %eax,%eax          # count = 0;
25         andl %edx,%edx          # len <= 0?
26         jle Done                # if so, goto Done:
27
28 Loop:   mrmovl (%ebx), %esi     # read val from src...
29         rmmovl %esi, (%ecx)     # ...and store it to dst
30         andl %esi, %esi         # val <= 0?
31         jle Npos                # if so, goto Npos:
32         irmovl $1, %edi
33         addl %edi, %eax         # count++
34 Npos:   irmovl $1, %edi
35         subl %edi, %edx         # len--
36         irmovl $4, %edi
37         addl %edi, %ebx         # src++
38         addl %edi, %ecx         # dst++
39         andl %edx,%edx          # len > 0?
40         jg Loop                 # if so, goto Loop:
41 ##################################################################
42 # Do not modify the following section of code
43 # Function epilogue.
44 Done:
45         popl %edi               # Restore callee-save registers
46         popl %ebx
47         popl %esi
48         rrmovl %ebp, %esp
49         popl %ebp
50         ret
51 ##################################################################
52 # Keep the following label at the end of your function
53 End:
```


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

- 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

```
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 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.ys` 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.ys` 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.ys -n K -rc > driver.ys
  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.ys` and `ldriver.ys`, 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, 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

Part A - Homework 5

Part A is worth 100 points broken down as follows:

- 10 points if all programs compile correctly (this is an all or nothing, if any one program does not compile correctly then no credit is received)
- 30 points for each program if the program functions correctly for the test case specified in writeup, so make sure to include the test case in your source code so all we have to do is run it to see if it works. There will be partial credit but consider the following mentioned below.

Your programs must be functionally equivalent to the C code in **figure 1**, this means that sum.ys and rsum.ys must work for any size linked list, and copy.ys must work for any size block depending on the parameter `len` (so to properly copy the block in our test input you should pass 3 as len). We will check to make sure your source code is not hard coded for the sizes of the test input we give in the write up or any other constant size.

For the data provided in the writeup, 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 `0xfed` 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 `0xfed` in register `%rax`, copies the three 64-bit values `0x00d`, `0x0e`, and `0xf` to the 24 bytes beginning at address `dest`, and does not corrupt other memory locations.

Part B - Homework 6

This part of the project is worth 100 points:

- 30 points for your description of the computations required for the `iaddq` instruction.

- 30 points for passing the benchmark regression tests in `y86-code`, to verify that your simulator still correctly executes the benchmark suite.
- 40 points for passing the regression tests in `ptest` for `iaddq`.

Part C - Project 2

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

- 20 points each for your descriptions in the headers of `ncopy.ys` 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 YIS, and `pipe-full.hcl` passes all tests in `y86-code` and `ptest`.

  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 project will be:

$$S = \begin{cases} 0\,, & c > 10.5 \\ 28.571 \cdot (10.5 - c)\,, & 8.40 \leq c \leq 10.50 \\ 60\,, & c < 8.40 \end{cases}$$

  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.

8 Submission Rules

We will attempt to use a grading script to make grading faster. This means that you should follow the format specified in each section, or there could be an error in your grading.

Part A

You must prefix the names of each of your files with **your netid**, make sure you use the '-' minus symbol to separate your netid and the rest of the file name. Here is an example of the names of someone's files.
`netid3-copy.ys`
`netid3-rsum.ys`
`netid3-sum.ys`
Then zip these three files together, make sure not to zip a directory containing these files, and submit the zip file to blackboard.

Part B

You must prefix the names of each of your files with **your netid**, make sure you use the '-' minus symbol to separate your netid and the rest of the file name. Here is an example of the name of someone's files.
`netid3-seq-full.hcl`
Just submit the hcl file to blackboard, no need to zip a single file.

Part C

You must prefix the names of each of your files with **your netid**, make sure you use the '-' minus symbol to separate your netid and the rest of the file name. Here is an example of the names of someone's files.
`netid3-ncopy.ys`
`netid3-pipe-full.hcl`
Then zip these two files together, make sure not to zip a directory containing these files, and submit the zip file to blackboard.

9 Hints

- By design, both `sdriver.yo` and `ldriver.yo` are small enough to debug with in GUI mode. We find it easiest to debug in GUI mode, and suggest that you use it.
- If you running in GUI mode on a Unix server, 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.