CS359, Spring 2017
Project2: Optimizing the Performance of a Pipelined Processor
Assigned: Mar. 30th, Due: Apr. 15th, 11:59PM

Late Policy: no late submission is accepted.

1 Introduction

In this project, you will learn about the design and implementation of a pipelined Y86 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 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 programs and become familiar with the Y86 tools. In Part B, you will extend the SEQ simulator with a new instructions. These two parts will prepare you for Part C, the heart of the project, where you will optimize the Y86 benchmark program and the processor design.

2 Logistics

You can make a team with 2-3 members to complete this project. Of course, you're allowed to work on this project alone if you prefer to do it yourself. For a team, please select one of the team members to be the corresponding leader, which makes it convenient for us to contact your team.

All handins must be electronic. Any necessary clarifications and revisions to the assignment will be posted on the course web page.

3 Handout Instructions

1. Start by copying the file `project2-handout.zip` to a (protected) directory on a Linux machine in which you plan to do your work as project1 does.
2. Then give the command: `unzip project2-handout.zip`. This will cause the following files to be unpacked into the directory: `README`, `Makefile`, `sim.zip`, `project2.pdf`, and `simguide.pdf`.

3. Next, give the command `unzip sim.zip`. This will create the directory `sim`, which contains your personal copy of the Y86 tools. You will be doing all of your work inside this directory.

4. Finally, change to the `sim` directory and build the Y86 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 programs. The required behavior of these programs is defined by the example C functions in `examples.c`. Be sure to put leader's name and student 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.

In all of your Y86 functions, you should follow the IA32 conventions for the structure of the stack frame and for register usage instructions, including saving and restoring any callee-save registers that you use.

`sum.ys`: Iteratively sum linked list elements

Write a Y86 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 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 4
ele1:
        .long 0x00a
        .long ele2
ele2:
        .long 0x0b0
        .long ele3
ele3:
        .long 0xc00
        .long 0
```

`rsum.ys`: Recursively sum linked list elements

Write a Y86 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`.

```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 solution functions.** See `sim/misc/examples.c`

Byte	1	2	3 4 5
iaddl V, rB	C 0	F rB	V

Table 1: Format of iaddl

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 4
# Source block
src:
        .long 0x00a
        .long 0x0b0
        .long 0xc00

# Destination block
dest:
        .long 0x111
        .long 0x222
        .long 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 a new instruction: iaddl. The function of iaddl is to add a constant value to a register. This requires first using an irmovl instruction to set a register to the constant, and then an addl instruction to add this value to the destination register. Suppose we want to add a new instruction iaddl with the format as illustrated in Table 1:

To add these instructions, you will modify the file seq-full.hcl, which implements the version of SEQ described in the CS:APP2e 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:

- Leader's name and ID.
- A description of the computations required for the iaddl instruction. Use the descriptions of irmovl and OPl in Figure 4.18 in the CS:APP2e 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 program.* For your initial testing, we recommend running simple programs such as `asumi.yo` (testing `iaddl`) 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 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 `iaddl`:

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

  To test your implementation of `iaddl`:

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

For more information on the SEQ simulator refer to the handout *CS:APP2e Guide to Y86 Processor Simulators* (`simguide.pdf`).

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

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

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 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 `IIADDL`.

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:

- Leader's 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.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.

```
##################################################################
# ncopy.ys - Copy a src block of len ints to dst.
# Return the number of positive ints (>0) contained in src.
#
# Include your name and ID here.
#
# Describe how and why you modified the baseline code.
#
##################################################################
# Do not modify this portion
# Function prologue.
ncopy:  pushl %ebp              # Save old frame pointer
        rrmovl %esp,%ebp        # Set up new frame pointer
        pushl %esi              # Save callee-save regs
        pushl %ebx
        pushl %edi
        mrmovl 8(%ebp),%ebx     # src
        mrmovl 16(%ebp),%edx    # len
        mrmovl 12(%ebp),%ecx    # dst

##################################################################
        # Loop header
        xorl %eax,%eax          # count = 0;
        iaddl $-1, %edx
        jle Npos3               # if so, goto Done:
Loop:
        mrmovl (%ebx), %esi     # read val from src...
        mrmovl 4(%ebx), %edi    # read val from src...
        rmmovl %esi, (%ecx)     # ...and store it to dst
        rmmovl %edi, 4(%ecx)    # ...and store it to dst
        andl %esi, %esi         # val <= 0?
        jle Npos1               # if so, goto Npos:
        iaddl $1, %eax          # count++
Npos1:
        andl %edi, %edi         # val <= 0?
        jle Npos2               # if so, goto Npos:
        iaddl $1, %eax          # count++
Npos2:
        iaddl $8, %ebx          # src++
        iaddl $8, %ecx          # dst++
        iaddl $-2, %edx         # len--
        jg Loop                 # if so, goto Loop:
Npos3:
        jne Done
        mrmovl (%ebx), %esi     # read val from src...
        rmmovl %esi, (%ecx)     # ...and store it to dst
        andl %esi, %esi         # val <= 0?
        jle Done                # if so, goto Npos:
        iaddl $1, %eax          # count++
##################################################################
# Do not modify the following section of code
# Function epilogue.
Done:
        popl %edi               # Restore callee-save registers
        popl %ebx
        popl %esi
        rrmovl %ebp, %esp
        popl %ebp
        ret
##################################################################
```

- Your `ncopy.ys` function must run correctly with YIS. By correctly, we mean that it must correctly copy the `src` block *and* return (in `%eax`) 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` flags that test `iaddl`).

Other than that, you are free to implement the `iaddl` 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:APP2e.

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 `%eax` 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 `%eax` 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 `%eax` 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 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 `iaddl` 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

The project is worth 250 points: 30 points for Part A, 60 points for Part B, 100 points for Part C, and 60 points for report.

Part A

Part A is worth 30 points, 10 points for each Y86 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 `%eax`.

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 `%eax`, copies the three words `0x00a`, `0x0b`, and `0xc` to the 12 contiguous memory locations beginning at address `dest`, and does not corrupt other memory locations.

Part B

This part of the project is worth 60 points:

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

Part C

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 914 cycles to copy 63 elements, for a CPE of $914/63 = 14.51$.

  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 46.0 and 14.51, with an average of 16.44. Note that this Perl script does not check for the correctness of the answer. Use the script `correctness.pl` for this:

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

  You should be able to achieve an average CPE of less than 10.0. Our best version averages 9.27. If your average CPE is c, then your score S for this portion of the project will be:

$$S = \begin{cases} 0\,, & c > 12.5 \\ 24.0 \cdot (12.5 - c)\,, & 10.0 \leq c \leq 12.5 \\ 60\,, & c < 10.0 \end{cases}$$

  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 Handin Instructions

- To make our evaluation efficiently, please submit your work as following instructions. For your team, assume the leader's student ID as 00000000, you will be handing in three sets of files:

- folder `parta`: `00000000-sum.ys`, `00000000-rsum.ys`, and `00000000-copy.ys`.
- folder `partb`: `00000000-seq-full.hcl`.
- folder `partc`: `00000000-ncopy.ys` and `00000000-pipe-full.hcl`.

Then please put the three folders and your report file with PDF format named as `00000000-report.pdf` into the final folder: `00000000` and zip it as `"00000000.zip"`.

- Make sure you have included leader's name and ID in a comment at the top of each of your handin files.

- To handin your files for part X, go to your `project2-handout` directory and type:

```
unix>  make handin-partX TEAM=teamname
```

where `X` is `a`, `b`, or `c`, and where `teamname` is leader's student ID. For example, to handin Part A:

```
unix>  make handin-parta TEAM=teamname
```

- Finally, please upload your zip file into the folder: `"/upload/ComputerArchitecture/Project2"` on `"ftp://public.sjtu.edu.cn"` via following account:

```
Account: zhouxian
Password: cscscscs
```

- *IMPORTANT:* Do add all the names and student IDs of all team members. The report is the only file that contains information of team members except for the leader. And for each member, an available email is also needed.

- *Requirements of report:* we recommend you to write the report with Latex because it do help you make a good typesetting. Microsoft office is also okay to make a PDF file. There are three parts: parta, partb, and partc. For each part, we'd like to see description, solution, analysis, outcome with some illustrations if necessary. At last, please state task assignments in your team. Some gain or thinking personally is also recommended.

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.

- 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 object file.

10 Last but not least

Notice that our late policy: *No late submission is accepted!*

Please contact cs_architecture@163.com if you meet any problem for this project. Good luck, everybody ☺.