CS 203: Optimizing the Performance of a Pipelined Processor

1   Introduction

In this project, you will learn about the design and implementation of a pipelined Y86 processor, optimizing its performance on a benchmark program. 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 two 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 may work in groups for this project.

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

3   Handout Instructions

1. Start by copying the file tar file (on webpage) to a directory in which you plan to do your work.

2. Then give the command: `tar xvf archlab-handout.tar`. This will cause the following files to be unpacked into the directory: `README`, `Makefile`, `sim.tar`, `archlab.ps`, `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 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 Schedule

Lab 7: `sum.ys` from part A should be submitted via Moodle by 11:55pm on Monday November 26th.

Lab 8: `iaddl` should be submitted (which is in `seq-full.hcl`) via Moodle by 11:55pm on Friday November 30th.

Project 2: The remainder of the project, along with `sum.ys` and `seq-full.hcl`, is due by 11:55pm on Friday Decemeber 7th.

5 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 your name and ID in a comment at the beginning of each program.

`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 a main routine that invokes a Y86 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
```

You can compile by doing someting like `yas sum.ys` and then simulating with something like `../seq/ssim -g sum.ys`, where the (optional) flag g says to use the gui. To get started, you will probably want to look at other .ys files, by doing a `find . | grep ys` to find all the .ys files that you just untar'ed.

`rsum.ys`: Recursively sum linked list elements

Write a recursive version of `sum.ys` (`rsum.ys`) that recursively sums the elements of a linked list.

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

Your program should consist of a main routine that invokes a recursive Y86 function (`rsum_list`) that is functionally equivalent to the `rsum_list` function in Figure 1. Test your program using the same three-element list you used for testing `list.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 a main routine that calls a Y86 function (`copy_block`) that is functionally equivalent to the `copy_block` function in 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
```

6   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 two new instructions: `iaddl` (described in homework problems 4.47 and 4.49) and `leave` (described in homework problems 4.48 and 4.50). To add these instructions, you will modify the file `seq-full.hcl`, which implements the version of SEQ described in the CS:APP 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

- A description of the computations required for the `iaddl` instruction. Use the descriptions of `irmovl` and `OPl` in Figure 4.18 in the CS:APP text as a guide.

- A description of the computations required for the `leave` instruction. Use the description of `popl` in Figure 4.20 in the CS:APP text as a guide.

4

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 login 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 a simple program such as `asum.yo` in TTY mode, comparing the results against the ISA simulation:

  ```
  unix>  ./ssim -t asum.yo
  ```

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

  ```
  unix>  ./ssim -g asum.yo
  ```

- *Testing 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. 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` and `leave`:

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

  To test your implementation of `iaddl`:

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

  To test your implementation of `leave`:

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

  To test both `iaddl` and `leave`:

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

For more information on the SEQ simulator refer to the handout *CS:APP Guide to Y86 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.

7 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:

- Your name

- 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         # Function prologue. Do not modify.
11 ncopy:  pushl %ebp               # Save old frame pointer
12         rrmovl %esp,%ebp         # Set up new frame pointer
13         pushl %esi               # Save callee-save regs
14         pushl %ebx
15         mrmovl 8(%ebp),%ebx      # src
16         mrmovl 12(%ebp),%ecx     # dst
17         mrmovl 16(%ebp),%edx     # len
18
19         # Loop header
20         xorl %esi,%esi           # count = 0;
21         andl %edx,%edx           # len <= 0?
22         jle Done                 # if so, goto Done:
23
24         # Loop body.
25 Loop:   mrmovl (%ebx), %eax      # read val from src...
26         rmmovl %eax, (%ecx)      # ...and store it to dst
27         andl %eax, %eax          # val <= 0?
28         jle Npos                 # if so, goto Npos:
29         irmovl $1, %edi
30         addl %edi, %esi          # count++
31 Npos:   irmovl $1, %edi
32         subl %edi, %edx          # len--
33         irmovl $4, %edi
34         addl %edi, %ebx          # src++
35         addl %edi, %ecx          # dst++
36         andl %edx,%edx           # len > 0?
37         jg Loop                  # if so, goto Loop:
38
39         # Function epilogue. Do not modify.
40 Done:   rrmovl %esi, %eax
41         popl %ebx
42         popl %esi
43         rrmovl %ebp, %esp
44         popl %ebp
45         ret
```

Figure 3: **Baseline Y86 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 `%eax`) the correct number of positive integers.

- Your `pipe-full.hcl` implementation must pass the regression tests in `../y86-code` and `../ptest` (without the `-il` flags that test `iaddl` and `leave`).

Other than that, you are free to implement the `iaddl` instruction if you think that will help. You are free to alter the branch prediction behavior or to implement techniques such as load bypassing. You may make any semantics preserving transformations to the `ncopy.ys` function, such as swapping instructions, replacing groups of instructions with single instructions, deleting some instructions, and adding other instructions.

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 3 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 62 (`0x3e`) 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 your modify your `pipe-full.hcl` file, you can rebuild the simulator by typing

```
unix>  make psim
```

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

```
unix>  make
```

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>  cd sim/pipe
  unix>  make
  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 1 up to some limit (default 64), simulates them with YIS, and checks the results. It generates a report showing the status for each block length:

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

  If you get incorrect results for some length K, you can generate a driver file for that length that includes checking code:

  ```
  unix>  ./gen-driver.pl -n K -c > driver.ys
  unix>  make driver.yo
  unix>  ../misc/yis driver.yo
  ```

  The program will end with register `%eax` having value `0xaaaa` if the correctness check passes, `0xeeee` if the count is wrong, and `0xffff` if the count is correct, but the words are not all copied correctly.

- *Testing your 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 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)
  ```

8 Evaluation

Lab 7

This lab is worth **10 points**. Each solution program will be evaluated for correctness, including proper handling of the `%ebp` stack frame register and functional equivalence with the example C functions in `examples.c`.

`sum.ys` will be considered correct if `sum_list` returns the sum `0xcba` in register `%eax`.

Lab 8

This lab is worth **20 points**. 10 points will be based on your description of the computations required for the `iaddl` instruction. 10 points will be based on passing the regression tests in `iaddl` and the benchmark regression tests in `y86-code`, to verify that your simulator still correctly executes the benchmark suite.

Project

The project will be worth **80 points**.

Part A

Part A is worth 10 points, 5 points for each Y86 solution program. Each solution program will be evaluated for correctness, including proper handling of the `%ebp` stack frame register and functional equivalence with the example C functions in `examples.c`.

`rsum.ys` will be considered correct if `rsum_list` returns the sum `0xcba` in register `%eax`.

The program `copy.ys` will be considered correct if its `copy_block` function returns the sum `0xcba` in register `%eax`, and copies the three words `0x00a`, `0x0b`, and `0xc` to the 12 contiguous memory locations beginning at address `dest`.

Part B

This part of the project is worth 20 points:

- 10 points for your description of the computations required for the `leave` instruction.

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

Part C

This part of the project is worth 50 points:

- 5 points each for your descriptions in the headers of `ncopy.ys` and `pipe-full.hcl`.

- 40 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 display 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 1037 cycles to copy 63 elements, for a CPE of $1037/63 = 16.46$.

  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 45.0 and 16.45, with an average of 18.15. 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 12.0. Our best version averages 7.43.

  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.

  The two best average CPEs in the class will receive two and one points **extra credit**, respectively, on the final.

9 Handin Instructions

- You will be handing in the following files:
  - Part A: `sum.ys`, `rsum.ys`, and `copy.ys`.
  - Part B: `seq-full.hcl`.
  - Part C: `ncopy.ys` and `pipe-full.hcl`.

- Make sure you have included your name in a comment at the top of each of your handin files.

- Please submit files via moodle when finished.

10   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 box, 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 object file.

- When running in GUI mode, the `psim` and `ssim` simulators will single-step past a `halt` instruction.