91.305 Computer Architecture, Fall 2003
Assignment 7: Introduction to the Y86 and SEQ
Out: Fri Oct 31, Due: Fri Nov 7

1 Introduction

In this lab, you will learn about the design and implementation of a the Y86 processor.

The lab is organized into two parts.

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 next week—the heart of the lab—where you will optimize the Y86 benchmark program and the processor design.)

2 Logistics

You will work on this lab alone.

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

3 Handout Instructions

1. Log into `mercury.cs.uml.edu`.
2. Start by copying the file `assignment7.tar`, from the course website, to a (protected) directory on `mercury` in which you plan to do your work. You can do this by saying: `cp ~fredm/305/files/assignment7.tar .`
3. Then give the command: `tar xvf assignment7.tar`. This will cause the following files to be unpacked into an `assignment7` directory: including `assignment7.pdf`

(this file), `simguide.pdf` (guide to the CMU simulators), and directories `sim/`, `misc/`, `ptest/`, and `y86-code/`.

4. 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/` in this part.

Your task is to write and simulate the following two Y86 programs. The required behavior of these programs is defined by the example C functions in `examples.c`. Be sure to put your name 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 be 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
```

Write your program in the file `sim/sum.ys`. I have already started you off in the right direction to iterate through the list, summing it up.

Test your code first by assembling the `sum.ys` file with the `yas` assembler:

```
unix>  yas sum.ys
```

`yas` lives in `/usr/local/bin/`. Make sure your `PATH` environment variable is set up properly, or you'll have to give the full path to `/usr/local/bin/yas` each time you need to run it.

Then, run the simulator by saying:

```
unix>  ./ssim sum.yo
```

This will produce output on the terminal showing what happened. For example, for the little getting-started code I've supplied you with in `sum.ys`, running the simulator produces:

```
Y86 Processor: seq-full.hcl
45 bytes of code read
IF: Fetched irmovl at 0x0.  ra=----, rb=%eax, valC = 0x0
IF: Fetched irmovl at 0x6.  ra=----, rb=%edx, valC = 0x18
IF: Fetched mrmovl at 0xc.  ra=%ebx, rb=%edx, valC = 0x0
IF: Fetched addl at 0x12.  ra=%ebx, rb=%eax, valC = 0x0
IF: Fetched halt at 0x14.  ra=----, rb=----, valC = 0x0
5 instructions executed
Exception status = HLT
Condition Codes: Z=0 S=0 O=0
Changed Register State:
%eax: 0x00000000 0x0000000a
%edx: 0x00000000 0x00000018
%ebx: 0x00000000 0x0000000a
Changed Memory State:
```

If you want to single-step through your program, then you need to be running an X server on your desktop.

You can do this by either:

1. Go to Olsen and use a Linux machine in OS308. After logging in, connect to Mercury by saying `ssh -X mercury`. Then, from the shell session, run the simulator with `./ssim -g sum.yo` (the `-g` means "graphics version"). The group of simulator windows should pop open on your screen.

2. Install an X Windows server on your own computer. The open-source Cygwin package has one. Also, if you are behind a NAT box or other firewall, you have to configure it to forward port 22 to your machine. If you don't know what this means, get help from someone who does, or pursue option 1 above.

   Then, from your X Windows environment, open a local XTerm. From this prompt, do the `ssh -X mercury.cs.uml.edu` and log in to Mercury. From Mercury, try saying `xeyes &`. If a pair of eyeballs shows up on your screen, X is working! Now you can run `./ssim -g sum.yo`.

Please don't wait until the last minute to configure yourself for working at home. If you are serious about getting this working, get your setup running with time to spare if you run into difficulty.

Turn in your assembled `sum.yo` file, plus a copy of the tty simulator results showing the correct sum in the `%eax` register. Note: you can say `./ssim sum.yo > sum.txt` to dump the simulator output into a file named `sum.txt`.

`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
```

I have gotten you started with a `copy.ys` source file.

Turn in your assembled `copy.yo` file, plus a copy of the tty simulator results showing how memory has changed (and the XOR'ed result in register `%ebx`).

5 Part B

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

Your task in Part B is to extend the SEQ processor to support two new instructions: `iaddl` (described in homework problems 4.32 and 4.34) and `leave` (described in homework problems 4.33

```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;
}

/* 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`

and 4.35). 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 4.16 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 4.18 in the CS:APP 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 edit the `Makefile` script assign `VERSION=full` there. (Then you can just say `make` to build a new version.)

- *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`).