Middle East Technical University Department of Computer Engineering

CENG 331

Computer Organization

Fall 2015-2016

Lab Exam 3

Prelab due date: December 21st, 2015, Monday, 17:40:00
Exam date: December 21st, 2015, Monday, 17:40:00

1 Objective

In this lab, you will learn about the design and implementation of a sequential Y86-64 processor.

The lab is organized into two 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.

For the study, we supplied a prelab exam which will also be evaluated.In lab-exam you will encounter similar problems.

Keywords: *Y86-64, SEQ, Architecture*

2 Prelab Specifications

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 will cause the following files to be unpacked into the directory: `README`, `Makefile`, `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
```

2.1 Part A

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

Your task is to write and simulate the following two 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 ID 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 0x00a
        .quad ele2
ele2:
        .quad 0x0b0
        .quad ele3
ele3:
        .quad 0xc00
        .quad 0
```

`rsum.ys`: Recursively sum linked list elements

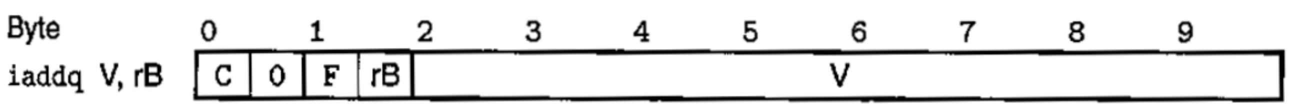
Write a Y86-64 program `rsum.ys` that recursively sums the elements of a linked list. This code should be similar to the code in `sum.ys`, except that it should use a function `rsum_list` that recursively sums a list of numbers, as shown with the C function `rsum_list` in Figure 1. Test your program using the same three-element list you used for testing `list.ys`.

2.2 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 below.

One common pattern in machine-level programs is to add a constant value to a register. With the Y86-64 instructions presented thus far, this requires first using an irmovq instruction to set a register to the constant, and then an addq instruction to add this value to the destination register. Suppose we want to add a new instruction iaddq with the following format:



This instruction adds the constant value V to register rB.

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 ID.
- 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 (See Figure 2) which is also provided as reference, as a guide.

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

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

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

```

Figure 1: C versions of the Y86-64 solution functions.

See `sim/misc/examples.c`

Stage	OPq rA, rB	rrmovq rA, rB	irmovq V, rB
Fetch	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1] valP $\leftarrow$ PC + 2	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1] valP $\leftarrow$ PC + 2	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1] valC $\leftarrow$ M_8[PC + 2] valP $\leftarrow$ PC + 10
Decode	valA $\leftarrow$ R[rA] valB $\leftarrow$ R[rB]	valA $\leftarrow$ R[rA]	
Execute	valE $\leftarrow$ valB OP valA Set CC	valE $\leftarrow$ 0 + valA	valE $\leftarrow$ 0 + valC
Memory			
Write back	R[rB] $\leftarrow$ valE	R[rB] $\leftarrow$ valE	R[rB] $\leftarrow$ valE
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ valP	PC $\leftarrow$ valP

Figure 2: 4.18 of the CS:APP3e text.

Computations in sequential implementation of Y86-64 instructions OPq, rrmovq, and irmovq. These instructions compute a value and store the result in a register. The notation icode : ifun indicates the two components of the instruction byte, while rA: rB indicates the ”two components of the register specifier byte. The notation M_1[x] indicates accessing (either reading or writing) 1 byte at memory location x, while M_8[x] indicates accessing 8 bytes.

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`).

3 Evaluation

The lab is worth 100 points: 30 points for Part A and 70 points for Part B.

Part A

Part A is worth 30 points, 15 points for each Y86-64 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 `%rax`.

Part B

This part of the lab is worth 70 points:

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

4 Submission

Submission will be done via COW. Create a tar file named `lab3.tar` that contains all your source code files using the following command where studentId is of the form e1234567

```
$ make submission TEAM=studentId
```

5 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.