4190.308 Computer Architecture, Spring 2019
Optimizing the Code Size of Y86-64 ISA
Assigned: Wed, Apr 17th. Due: Mon, May 13th, 23:59 AoE

1 Introduction

Evolution in semiconductor technology has allowed the birth of sophisticated computing hardware. This gave advantage to our commodity PCs to have fast and abundant memory resources. Nevertheless, embedded systems developers need to consider the cost-sensitive or memory constrained hardware to store the full image of the binary executable. As a consequence, any optimization in code size translates directly to savings in cache size, which leads to reduction in unit cost of the chip. In this lab, you will learn about the design and implementation of a pipelined Y86-64 ISA by adding new instructions to it and modifying its implementation to minimize the size of the code, so that Y86-64 based embedded systems can be cost efficient.

The lab is organized into three parts. In the Part A, you will write some simple Y86-64 programs and become familiar with the Y86-64 tools. In Part B, you extend the assembler to support new instructions. These two parts will prepare you for Part C, where you will modify the SEQ/PIPE simulators.

2 Logistics and Handout Instructions

You will work on this lab alone. Any clarifications and revisions to the assignment will be posted on the course Web page. This time, we distribute the directory structure and source code of the lab through git server, at `https://git.csap.snu.ac.kr`

1. First press `fork` from the repository `comparchTA/Processor Lab`. This will create an identical repository on your git account.
2. If you have successfully forked the project, you would have your project on `https://git.csap.snu.ac.kr/<your_id>/processor-lab`. Please make sure that your repository is private by checking your option at `Settings > Permissions > Project visibility`.
3. Go to your project webpage, and press `clone`. You will get your remote repository URLs in HTTPS. Copy any address and clone the repository under your VM by typing this line on your Terminal:

```
$>  git clone https://<URL>
```

4. There are 3 directories in the root directory. You will find the Y86-64 tools in the `sim` directory. `parta` is the place where you will submit 3 `.ys` files from Part A. In `report`, you will provide us with your report for this lab.

5. Finally, change to the `sim` directory and build the Y86-64 tools:

   ```
   $>  cd sim
   $>  make clean; make
   ```

   Note: you may get a long list of warnings (`...warning:  'result' is deprecated ...`). Those are not a problem and can be ignored.

3 Part A

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 `sim/misc/examples.c` shown in Figure 1. You can test your programs by first assembling them with the program `yas` and then running them with the instruction set simulator `yis`. Both programs are located in the directory `sim/misc`.

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. You may want use the Y86-64 program skeleton from the lecture slides as a starting point of your implementation.

`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 (lines 7–16). 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
```

```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;
    }
}

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

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

`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 (lines 18–28). 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 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 (lines 30–41). Test your program using the following three-element source and destination blocks:

```
.align 8
# Source block
src:
        .quad 0x00a
        .quad 0x0b0
        .quad 0xc00

# Destination block
dest:
        .quad 0x111
        .quad 0x222
        .quad 0x333
```

4 Part B

Your task in Part B is to improve the code size of Y86-64 by extending new instructions available to the Y86-64 compiler. Currently, `irmovq` and `call` only accept immediate values as signed inputs that have a fixed size of 8 bytes. For `irmovX`, our new instructions must allow different immediate sizes. For `call`, our new instruction uses PC-relative offsets to reduce the number of bytes needed to represent the address. For example:

```
0x000:  irmovq stack,%rsp           # rsp = address of stack (0x50)

0x00a:  irmovq $0x123,%rax          # rax = 0x0000000000000123
0x014:  irmovl $0x11223344,%rcx     # rcx = 0x11223344
0x01a:  irmovw $0xffde,%rdx         # rdx = -0x22
0x01e:  irmovb $ff,%rbx             # rbx = -0x1

0x021:  callo test                  # call test by relative offset (0x21 + 0x10)
0x026:  halt                        # halt program
```

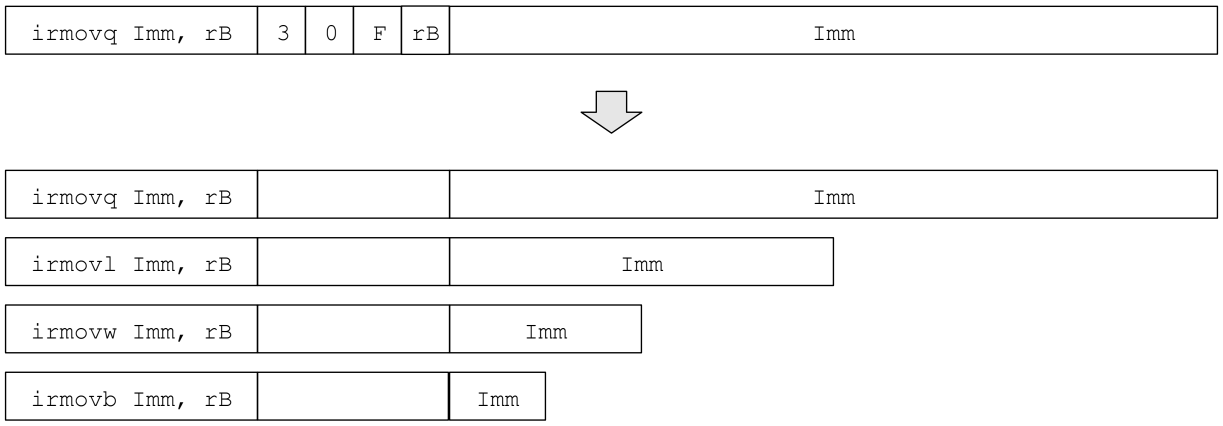

Figure 2: Instruction encoding for the new `irmovX` that supports different immediate sizes

```
0x027:  irmovq $6,%rsi              # Never executed

0x031:   test:
0x031:   irmovq $-5,%rax             # rax = -5
0x03b:   ret                         # return to original code
0x03c:   nop                         # dummy

0x050:   .pos 0x50:
0x050:   stack:
```

By extending the instruction set encoding, we get four new instructions: `irmovl`, `irmovw`, `irmovb`, and `callo`. One important restriction is that you must re-use the format of `irmovq` when creating `irmovX`. Think about what part of the instruction encoding you can use as an identifier to signal the type of the `irmovq`. Also, `irmovl`, `irmovw`, `irmovb` must handle signed immediate values of 4 bytes, 2 bytes, 1 byte, respectively.

For `callo`, the behavior is similar as `call`, which pushes the return address on the stack and jumps to the destination address. However, you may have to add a new instruction and operand type to handle the PC-relative offset. The immediate size of `callo` should be a signed 4-byte value.

4.1 Extending the Instruction Set

To extend the instruction set accepted by the processor, you will be working in the `sim/misc` directory. You have to add the new mnemonics to `yas-grammar.lex`, and then modify the instruction set definitions in `sim/misc/isa.c`. Additionally, in `isa.c`, the function that returns the mnemonic of the instruction will need to be fixed. Another change will be necessary in the function `step_state` that executes one instruction. Currently, `step_state` neither considers different immediate operands size nor PC-relative offset. Modify the code there slightly to ensure the immediate value is correctly decoded in the `FETCH` stage. Lastly, fix `yas.c` so that the assembler calculates the relative offset. To support PC-relative offsets, you may need to add an additional argument type that is treated as an offset and not an absolute immediate value.

Once you have completed this step, you can build a new assembler and instruction set simulator by executing `make` in the current directory. If everything goes well, you will now have two executables, `yas` and `yis` that assemble and simulate the extended Y86-64 processor. Test your assembler and simulator thoroughly with the new instructions. A few test examples and the expected code and results are provided in `y86-code/`. Make sure the instruction encoding is correct and also that the instruction mnemonics are displayed correctly. Also, please don't forget to comment your code.

4.2 Building the Assembler

To build and test your assembler, go through the following steps in the working directory of your assembler (`sim/misc`).

- *Building your assembler.* You can use `make` to build the assembler:

```
$>  make
```

- *Generating machine code with assembler.* You can generate machine code (`.yo`) by giving assembly file (`.ys`) to the assembler `yas`:

```
$>  ./yas ../y86-code/<name>.ys
```

  `sim/misc/README` file for more details.

5 Part C

For this task, you will be working in the `sim/seq` and `sim/pipe` directory and edit the file `seq-std.hcl` and `pipe-std.hcl`.

In this part, we want to extend the sequential Y86-64 processor simulator to support the `irmovX` and `callo` instructions. Carefully read through the `hcl` files. You will find the definitions of all the signals of the sequential and pipelined Y86-64 processor. Some of these signals need to be changed to support our new instructions. Make sure to comment your code

5.1 Building and Testing Your Simulators

To build and test your sequential/pipelined simulators, go through the following steps in the working directory of your simulator (`sim/seq` or `sim/pipe`).

- *Building dependencies.* Please refer to 4.2
- *Building a new simulator.* You can use `make` to build a new simulator:

```
$>  make
```

- *Testing your solution on a simple Y86-64 program.* For your initial testing, we recommend running simple programs such as `test1.yo` in TTY mode, comparing the results against the ISA simulation:

  ```
  $>  ./ssim -t ../y86-code/test1.yo
  $>  ./psim -t ../y86-code/test1.yo
  ```

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

  ```
  $>  ./ssim -g ../y86-code/test1.yo
  $>  ./psim -g ../y86-code/test1.yo
  ```

  `../y86-code/README` file for more details.

For more information on the SEQ simulator refer to the handout *CS:APP3e Guide to Y86-64 Processor Simulators* (`simguide.pdf`).

6 Evaluation

The lab is worth 100 points: 15 points for Part A, 35 points for Part B, 20 points for Part C, and 30 points for your report.

Part A

Part A is worth 15 points, 5 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`.

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 `%rax`, copies the three 64-bit values `0x00a`, `0x0b`, and `0xc` to the 24 bytes beginning at address `dest`, and does not corrupt other memory locations.

Part B

This part of the lab is worth 35 points:

- 20 points for your implementation of the computation required for the `irmovX` instruction.
- 15 points for implementing the `callo` instruction.

Part C

This part of the Lab is worth 20 points:

- 5 points each for your implementation of `irmovX` in `SEQ/PIPE` simulator respectively.
- 5 points each for your implementation of `callo` in `SEQ/PIPE` simulator respectively.

Report

Your report is worth 30 points. We will look your report in detail:

- Your report should not be longer than 6 pages (excluding the cover page).
- Avoid copy-pasting screenshots of your code. We have your code. What we ask for here is your thought process applied to solve the lab. (More of a general remark since you are not submitted code but the idea remains)
- You can also attach some diagrams to depict your implementation, if necessary.
- Delete italic text when submitting your report. Those are only guidelines to help you
- The name of the file must match `201X-XXXXX_processorlab_report.pdf` and placed under `report` directory.

7 Handin Instructions

Your source code among your report should be pushed to your `git` branches until the deadline. The timestamp of your final commit will be considered as the submission date. If you **fail** to follow the submission guidelines, your score may be a subject of reduction.

- You will at least commit changes for following four sets of files:
  - Part A: `sum.ys`, `rsum.ys`, and `copy.ys`. (under `parta` directory)
  - Part B: `yas-grammar.lex`, `isa.h`, `isa.c`, and `yas.c`.
  - Part C: `seq-std.hcl`, `ssim.c`, `pipe-std.hcl`, and `psim.c`.
  - Report: `201X-XXXX_processorlab_report.pdf` (under `report` directory)
- You commit and push your files,
  1. Make any modifications on your files.
  2. Check which files were changed by:
     ```
     $>  git status
     ```
  3. Add to the commit target by:

```
$>  git add <path/name of file>
```

4. Write commit message by:

```
$>  git commit
```

5. Push to the git server by:

```
$>  git push origin master
```

6. You are recommended to commit and push regularly to give yourself a chance to recover from your potential mistake, such as accidently overwriting or deleting files.

8 Hints

- In order to run in GUI mode the TK and TCL libraries are required. In the provided VM, all necessary libraries have been installed. However, if you choose to work on another system, you may have to install the TCL/TK libraries. In gentoo Linux, both libraries can be installed as follows:

```
$>  sudo emerge -a dev-lang/tcl dev-lang/tk
```