4190.308 Computer Architecture, Spring 2017

Optimizing the Performance of a Pipelined Processor

Due: (online) Sun., May 14, 14:00, (offline) Tue., May 16, 14:00

1 Introduction

In this lab, you will learn about the design and implementation of a pipelined Y86-64 processor by adding new instructions to it and modifying its implementation to maximize the performance of existing programs. You are allowed to make any enhancements to the pipelined processor, but the program must not be modified. When you have completed the lab, you will have a keen appreciation for the interactions between code and hardware that affect the performance of your programs.

The lab is organized into three parts. In the Warmup Part, you will write some simple Y86-64 programs and become familiar with the Y86-64 tools. In Part A, you extend both the ISA and SEQ simulators with new instructions. These two parts will prepare you for Part B, the heart of the lab, where you will optimize a pipelined version of the Y86-64 processor to run the given benchmark program as fast as possible.

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. As with the previous lab, we provide the tarball of the lab, `processorlab-handout.tgz`, on the course website (eTL).

1. First copy the file `processorlab-handout.tgz` to a directory in which you plan to do your work.
2. Unpack the tarball with: `tar xvzf processorlab-handout.tgz`. This will unpack the following four files: `README`, `Makefile`, `sim.tgz`, `processorlab.pdf`, and `simguide.pdf`.
3. Next, execute the command `tar xvzf sim.tgz`. 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:

```
$>  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 Warmup

The warmup task is not mandatory and is not included in the handin. However, we strongly recommend solving the following problems to become familiar with the Y86-64 instruction set and the simulation tools.

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 `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 `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
```

`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 `sum.ys`.

```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`

`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 A

Your task in Part A is to improve the Y86-64 processor by extending the functionality of the `OPq` instruction group to also accept immediate operands as their first operand such as, for example,

```
   iaddq $0x15, %rdx          # rdx = rdx + 0x15
   isubq $0xff, %rax          # rax = rax - 0x15
```

By extending the instruction set encoding, we get four new instructions: `iaddq`, `isubq`, `iandq`, and `ixorq` forming the `iOPq` group. One important restriction is that you have to use the same `icode:ifun` encoding in the first byte as the register-to-register `OPq` instructions for the four new instructions (see figure 2). Think about what part of the instruction encoding you can use as an identifier to signals whether or not the instruction is of type `OPq` or `iOPq`.

4.1 Extending the Instruction Set

The first step is to extend the instruction set accepted by the processor. For this part, 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 `misc/isa.c`. Additionally, in `isa.c`, the function that returns the mnemonic of the instruction will need to be fixed because it currently only considers the `icode:ifun` part of the instruction. Another change will be necessary in the function `step_state` that executes one instruction. Currently, `step_state` does not know about ALU instructions with immediate operands. Modify the code there slightly to ensure the immediate value is correctly decoded in the `FETCH` stage.

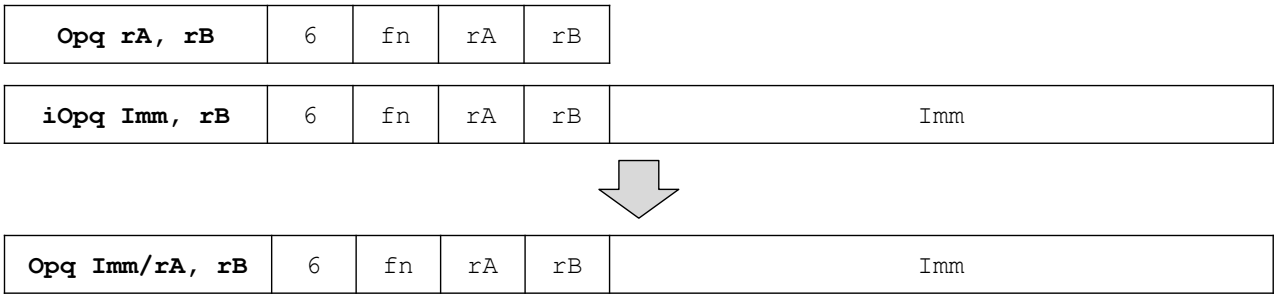

Figure 2: Instruction encoding for the new `OPq` that supports both immediate and register operands.

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 executable, `yas` and `yis` that assemble and simulate the extended Y86-64 processor. Test your assembler and simulator thoroughly with the new instructions. Make sure the instruction encoding is correct and also that the instruction mnemonics are displayed correctly.

4.2 Extending `SEQ` to Support `OPq` with Immediate Operands

Next, we want to extend the sequential Y86-64 processor simulator to support the `iOPq` instruction group. For this task, you will be working in the `sim/seq` directory and edit the file `seq-full.hcl`.

Carefully read through `seq-full.hcl`. You will find the definitions of all the signals of the sequential Y86-64 processor. Some of these signals need to be changed to support `iOPq` operations. For example, the signal `valC`, denoting whether the instruction contains an 8-byte constant or not, will need to be modified to account for the fact that under certain circumstances the `OPq` requires an immediate value.

Detailed instructions for building and testing your solution are given below in Section 4.4.

4.3 Extending `PIPE` to Support `OPq` with Immediate Operands

Last, we extend the pipelined version of the Y86-64 processor simulator to support `OPq` with immediate operands. For this task, you will be working in the `sim/pipe` directory and edit the file `pipe-full.hcl`.

Again, carefully read through `pipe-full.hcl`. A few of these signals need to be changed to support `OPq` operations with immediate operands. For example, also in the pipelined version the signal `valC` will need to be modified to support immediate operands.

Detailed instructions for building and testing your solution are given below in Section 4.4.

4.4 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 a new simulator.* You can use `make` to build a new simulator:

  ```
  $>  make VERSION=full
  ```

  This builds a version of `ssim/psim` that uses the control logic you specified in `seq/pipe-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:

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

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

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

  ```
  $>  (cd ../y86-code; make testssim)
  ```

  This will run your simulator 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 immediate operands run:

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

  (substitute `seq/ssim` by `pipe/psim` to test the pipelined version.) To test your implementation of `OPq` with immediate operands, run

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

  (again, substitute `seq/ssim` by `pipe/psim` to test the pipelined version of your simulator.)

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

```
1 /*
2  * ncopy - copy src to dst, returning number of positive ints
3  * contained in src array.
4  */
5 word_t ncopy(word_t *src, word_t *dst, word_t len)
6 {
7     word_t count = 0;
8     word_t 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 3: **C version of the `ncopy` function.** See `sim/pipe/ncopy.c`.

5 Part B

You will be working in directory `sim/pipe` in this part, and your task in is to modify `pipe-full.hcl` into a new file `pipe-full-optimized.hcl` and your goal is to increase the processor throughput. You can start off by testing the given example program `ncopy.ys`

Note that we will use our own set of test programs to grade your submission so make sure to test with other target programs as well.

The `ncopy` function in Figure 3 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 4 shows the baseline Y86-64 version of `ncopy`.

You will be handing in one file in this phase: `pipe-full-optimized.hcl`. However, we have added four extra files with different headers in them to guide you on some of the optimizations you might want to implement:

- `pipe-lf.hcl`: implements load forwarding logic.
- `pipe-nobypass.hcl`: `PIPE` without bypassing.
- `pipe-nt.hcl`: implements not taken strategy.
- `pipe-btfnt.hcl`: implements back-taken forward-not-taken strategy.

```
1  ##################################################################
2  # ncopy.ys - Copy a src block of len words to dst.
3  # Return the number of positive words (>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 # Do not modify this portion
11 # Function prologue.
12 # %rdi = src, %rsi = dst, %rdx = len
13 ncopy:
14
15 ##################################################################
16 # You can modify this portion
17         # Loop header
18         xorq %rax,%rax          # count = 0;
19         andq %rdx,%rdx          # len <= 0?
20         jle Done                # if so, goto Done:
21
22 Loop:   mrmovq (%rdi), %r10     # read val from src...
23         rmmovq %r10, (%rsi)     # ...and store it to dst
24         andq %r10, %r10         # val <= 0?
25         jle Npos                # if so, goto Npos:
26         irmovq $1, %r10
27         addq %r10, %rax         # count++
28 Npos:   irmovq $1, %r10
29         subq %r10, %rdx         # len--
30         irmovq $8, %r10
31         addq %r10, %rdi         # src++
32         addq %r10, %rsi         # dst++
33         andq %rdx,%rdx          # len > 0?
34         jg Loop                 # if so, goto Loop:
35 ##################################################################
36 # Do not modify the following section of code
37 # Function epilogue.
38 Done:
39         ret
40 ##################################################################
41 # Keep the following label at the end of your function
42 End:
```

Figure 4: **Baseline Y86-64 version of the `ncopy` function.** See `sim/pipe/ncopy.ys`.

Coding Rules

You are free to make any modifications you wish, with the following constraints:

- Your `pipe-full-optimized.hcl` implementation must pass the regression tests in `../y86-code` and `../ptest` (without the `-i` flag that tests `iOPq`).

Other than that, you are free to implement the `iOPq` instruction if you think that will help.

Building and Running Your Solution

In order to test your solution, we have provided four different driver programs that calls your `ncopy` function:

- `sdriver.ys`: 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 `%rax` after copying the `src` array.
- `ldriver.ys`: 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 `%rax` after copying the `src` array.
- `isdriver.ys`: Similar to `sdriver.ys` but with added `iOPq` instructions.
- `ildriver.ys`: Similar to `ldriver.ys` but with added `iOPq` instructions.

Each time you modify your `pipe-full-optimized.hcl` file, you can rebuild the simulator by typing.

```
$>  make psim VERSION=optimized
```

If you want to rebuild the simulator, type

```
$>  make VERSION=full
```

To test your solution in GUI mode on a small 4-element array, type

```
$>  ./psim -g sdriver.yo
$>  ./psim -g isdriver.yo
```

To test your solution on a larger 63-element array, type

```
$>  ./psim -g ldriver.yo
$>  ./psim -g ildriver.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 program files on the ISA simulator.* Make sure that your `ncopy.ys` function works properly with

  YIS:

  ```
  $>  ../misc/yis sdriver.yo
  ```

- *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 Y86-64 benchmark programs in `../y86-code`:

  ```
  $>  (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 `iOPq` instructions, then

  ```
  $>  (cd ../ptest; make SIM=../pipe/psim TFLAGS=-i)
  ```

6 Evaluation

The lab is worth 100 points: 40 points for Part A, 30 points for part B, and 30 points for the report.

Part A

Part A is worth 40 points, 20 points for extending the instruction set, 10 points for extending the `SEQ` hardware simulator to support `OPq` with immediate operands, and 10 points for extending the `PIPE` hardware simulator to support `iOPq`.

Part B

This part of the lab is worth 30 points and will be computed depending on the performance and correctness of your `pipe-full-optimized.hcl` implementation with our own test programs. Note that your implementation should pass all benchmark regression tests in `y86-code` and `ptest`, to verify that your simulator still correctly executes the benchmark suite.

We will express the performance of your implementation in units of *cycles per instruction* (CPI). The PIPE simulator displays the total number of cycles required to complete the program at the end of the simulation.

Report

Similar to the previous lab you are required to write a report about your implementation of the processor lab. You must include details on how you solved each one of the phases and what target programs you used to test the performance of your pipelined processor. Remember that even if you cannot solve part of the lab you need to write the section where you found some problem and couldn't progress, and what you tried to do in order to solve the problem.

A good rule of thumb in order to structure your report is to follow the sections mentioned in this handout (i.e., sections 4.1, 4.2, 4.3, and 5).

7 Handin Instructions

A `tar` file including the handin files of parts a, and b should be sent to comparch@csap.snu.ac.kr using the subject template: `[processor-lab][student-id]`.

Example: `[processor-lab][2016-00000]`.

Writing the subject of the email as we have specified is important as we will automate the process of sending a confirmation email once we have received your submission.

If you **fail** to follow the email guidelines you score will be subject of reduction.

- You will be handing in three sets of files:
  - Part A: `isa.c`, `yas-grammar.tex`, `seq-full.hcl` and `pipe-full.hcl`.
  - Part B: `pipe-full-optimized.hcl`.
- Make sure you have included your name and ID in a comment at the top of each of your handin files.
- To handin all parts, go to your `processorlab-handout` directory and type:

```
$>  make handin ID=0000-00000
```

- Similarly you can also create the handin files for specific parts, for partX, go to your `processorlab-handout` directory and type:

```
$>  make handin-partX ID=0000-00000
```

  where `X` is `a`, or `b`, and where `ID` is your student-ID. For example, to handin Part A:

```
$>  make handin-parta ID=2016-00000
```

8 Hints

- In order to run in GUI mode the TK and TCL libraries are required. Both libraries can be installed in the Gentoo (the provided VM) as follows:

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

  This might differ if you are using any other Linux distribution.

  Note that if you run into problems while installing **any** application on the provided VM it is because to reduce the size we removed some files used by package manager. To fix this you can run:

  ```
  $>  sudo emerge --sync
  ```

- 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:

  ```
  $>  setenv DISPLAY local.host:0
  ```

- If you are accessing a remote machine using `ssh`, you can also use the GUI by using the `-Y` flag to the `ssh` command. This requires that the remote machine allows forwarding the X-server (see `ssh` config file at `/etc/ssh/sshd_config`)
- 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.
- If you want to add your own personal test files to the set of `y86-code` benchmark you can do this by first placing your test program(s) in the `y86-code/` directory, and then adding the correct name in the `Makefile`.

  For example lets add `my_test.ys` to the benchmark. We modify 3 lines of the `Makefile` to include our new program in the lists `YOFILES`, `PIPEFILES`, and `SEQFILES`.

  ```
  YOFILES = abs-asum-cmove.yo . . . . my_test.yo
  PIPEFILES = asum.pipe  . . .  my_test.pipe
  SEQFILES = asum.seq  . . .  my_test.seq
  ```

Now, our `my_test.ys` file is included whenever you run:

```
$>  (cd ../y86-code; make testssim
$>  (cd ../y86-code; make testpsim
```