# Datorarkitektur, VT 2009
# Lab 3: Optimizing the Performance of a Pipelined Processor

Inge Frick

2009-02-18

Inge Frick (`inge@nada.kth.se`) is the lead person for this assignment.

## 1 Introduction

In this lab, 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 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, 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 lab, where you will optimize the Y86 benchmark program and the processor design.

## 2 Logistics

You will work on this lab either alone or in a group of two.

This assignement will be done on any of CSCs Unix-computers. You can probably also do it at home, but for the graphical simulator you need to have Tcl-Tk and you will have to change some Makefiles.

To convert files from prog.c to prog.ys you can compile prog.c on assembler.nada.kth.se with gcc -O2 -S prog.c and then by hand convert the generated prog.s to prog.ys. It's important to have the -O2 flag. If you do, the generated code will use registers for local variables. This will be much easier to convert to Y86 code.

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

## 3 Handout Instructions

1. Start by copying the file `/info/maskin09/labbar/lab3/archlab-handout.tar` to a directory (call it `lab3`) 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` and `sim.tar`.

3. Now change the team name at the beginning of `Makefile`.

4. 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.

5. Finally, 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/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(s) and ID(s) 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
```

### `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.

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.

```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;
}

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

/* 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`

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. See also pages 265-268 in the book for a similar example. 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
```

## 5 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 4.32 and 4.34) and `leave` (described in homework problems 4.33 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(s) and CSC user ID(s).
- 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 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`).

## 6 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(s) and CSC user ID(s).
- A high-level description of your code. In each case, describe how and why you modified your code.

```c
/*
 * ncopy - copy src to dst, returning number of positive ints
 * contained in src array.
 */
int ncopy(int *src, int *dst, int len)
{
    int count = 0;
    int val;

    while (len > 0) {
        val = *src++;
        *dst++ = val;
        if (val > 0)
            count++;
        len--;
    }
    return count;
}
```

Figure 2: **C version of the ncopy function.** See `sim/pipe/ncopy.c`.

### 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. Your code must be correct for arrays with more than 64 elements but we will only test for speed on arrays with 64 elements or less.
- 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` with the `-il` flags that test `iaddl` and/or `leave` if those instructions are implemented.

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 forwarding. 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:

```
##################################################################
# ncopy.ys - Copy a src block of len ints to dst.
# Return the number of positive ints (>0) contained in src.
#
# Include your name and ID here.
#
# Describe how and why you modified the baseline code.
#
##################################################################
        # You may change this code anyway you want but remember that
        # registers %ebx, %esi, %edi, %ebp and %esp must have the same
        # values at the end of the function as they had att the beginning.

ncopy:  pushl %ebp              # Save old frame pointer
        rrmovl %esp,%ebp        # Set up new frame pointer
        pushl %esi              # Save callee-save regs
        pushl %ebx
        mrmovl 8(%ebp),%ebx     # src
        mrmovl 12(%ebp),%ecx    # dst
        mrmovl 16(%ebp),%edx    # len

        # Loop header
        xorl %eax,%eax          # count = 0;
        andl %edx,%edx          # len <= 0?
        jle Done                # if so, goto Done:

        # Loop body.
Loop:   mrmovl (%ebx), %esi     # read val from src...
        rmmovl %esi, (%ecx)     # ...and store it to dst
        andl %esi, %esi         # val <= 0?
        jle Npos                # if so, goto Npos:
        irmovl $1, %esi
        addl %esi, %eax         # count++
Npos:   irmovl $1, %esi
        subl %esi, %edx         # len--
        irmovl $4, %esi
        addl %esi, %ebx         # src++
        addl %esi, %ecx         # dst++
        andl %edx,%edx          # len > 0?
        jg Loop                 # if so, goto Loop:

Done:
        popl %ebx
        popl %esi
        rrmovl %ebp, %esp
        popl %ebp
        ret
```

Figure 3: **Baseline Y86 version of the ncopy function.** See `sim/pipe/ncopy.ys`.

- `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 you 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 70), 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 -r > 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, `0xffff` if the count is correct, but the words are not all copied correctly and `0xbbxx` if a register that must be saved is no returned with its original value: `0xbbaa` for `%ebp`, `0xbbbb` for `%ebx`, `0xbbcc` for `%esi` and `0xbbdd` for `%edi`.

- *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)
  ```

## 7 Evaluation

The lab is worth 120 points: 15 points for Part A, 25 points for Part B, and 80 points for Part C.

Based on these points you will get a grade on this work. 78 points will give grade E, 88 points will give grade D, 98 points will give grade C, 108 points will give grade B and 117 points will give grade A.

### Part A

Part A is worth 15 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`.

The programs `sum.ys` and `rsum.ys` will be considered correct if their respective `sum_list` and `rsum_list` functions return 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 lab is worth 25 points:

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

**Part C**

This part of the Lab is worth 80 points:

- 20 points for your descriptions in the headers of `ncopy.ys` and `pipe-full.hcl`.
- 60 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 $6.98$.

  Performance is calculated by $max(0, min(60, 6 * (17 - x)))$ where x is the average CPE.

  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.

# 8 Handin Instructions

- You will be handing in three groups of 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(s) and CSC user ID(s) in a comment at the top of each of your handin files.
- To handin your files for part X, go to your `lab3` directory and type:

  ```
  unix>  make handin-partX
  ```

  where `X` is `a`, `b`, or `c`. For example, to handin Part A:

```
unix>  make handin-parta
```

- After the handin, if you discover a mistake and want to submit a revised copy, type

  ```
  unix  make handin-partX VERSION=2
  ```

  Keep incrementing the version number with each submission.

- You can verify your handin by looking in

  ```
  /info/maskin09/labbar/lab3/handin/partX
  ```

  You have list and insert permissions in this directory, but no read or write permissions.

## 9 Hints

- For part A see lecture 7A pages 13-20. See also files `asum.ys` and `asumr.ys` in `sim/y86-code`.
- For part B see lecture 8 and CS:app problems 4.32 - 4.35.
- For part C see lecture 9, 12, CS:app 4.5 and CS:app 5.
- Some ideas on how to speed up execution in part C:
  - Implement iaddl, see part B. I'ts important to test your implementation the same way you did in part B.
  - The machine expects branches to be taken. Arrange your code so this is true in most cases. Notice that most array elements are positive.
  - Do loop unrolling. Make it work doing e.g. 4 elements per round in the loop. For the last $n$ elements ($n < 4$) use a jumptable to enter the loop at the right place (see CS:app 3.6.6). When it works for 4 elements per round extend to more elements per round.
  - There is a load-use bubble in the code. You can get rid of the bubble by using load-forwarding (see CS:app problem 4.41 and pipe-lf.hcl) or by first fetching two elements from the source and then moving them to the destination.
- 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 are 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.