# CSCI 2021, Spring 2015
# Optimizing the Performance of a Pipelined Processor
# Assigned: March 23, Due: April 3, 11:55PM

Kartik Ramkrishnan (ramkr004@umn.edu) and Albert Jonathan (jonat004@umn.edu) are the lead persons for this assignment.

## 1 Introduction

In this lab, you will learn about the design and implementation of a pipelined Y86 processor, optimizing both it and a benchmark program to maximize performance. You are allowed to make any semantics preserving transformations to the benchmark program. 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 both the SEQ (sequential) and PIPE (pipeline) simulators with some new instructions. These two parts will prepare you for Part C, 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 Moodle.

## 3 Handout Instructions

1. Start by copying the file `archlab-handout.tar` to a (protected) directory in which you plan to do your work.
2. Then type in the command: `tar -xvf archlab-handout.tar`. This will cause the following file to be unpacked into the directory: `sim.tar, archlab.pdf, simguide.pdf, Makefile`.

3. Next, type in 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.

4. 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 two Y86 programs. The required behavior of these programs is defined by the example C functions in `examples.c`. You can test your programs by first assembling them with the program YAS and then running them with the instruction set simulator YIS.

In all of your Y86 functions, you should follow the IA32 conventions for the structure of the stack frame and for register usage instructions, including saving and restoring any callee-save registers that you use.

### `search.ys`: Search in a binary search tree

Write a Y86 program `search.ys` that recursively searches for an element in a binary search tree. Your program should consist of code segments that set up the stack structure, invoke a function, and then halt. In this case, the function should be Y86 code for a function (`search`) that is functionally equivalent to the C `search` function in Figure 1. Your code should work on tree of any size, including an empty tree.

### `matrix_xor.ys`: Perform a matrix XOR

Write a Y86 program `matrix_xor.ys` that performs a sequence of XOR operations on two matrices and stores the result into another matrix. Your matrix XOR function should take three matrices: `A[i][j]`, `B[i][j]` and `C[i][j]`. The `C[i][j]` is the sum of the pairwise XOR of row `i` of `A` with column `j` of `B`. It is similar to a matrix multiplication function with XOR instead of multiplication. Your program should consist of code segments that set up the stack structure, invoke a function, and then halt. In this case, the function should be Y86 code for a function (`matrix_xor.ys`) that is functionally equivalent to the C `matrix_xor` function in Figure 1.

# 5 Part B

You will be working in directory `sim/seq`, `sim/pipe` and `sim/misc` in this part.

Your task in Part B is to extend the SEQ and the PIPE processors to support five new leal instructions: `leal0`, `leal1`, `leal2`, `leal4` and `leal8`. The description of the `lealX` instruction is as follow:

```c
/* Tree element */
typedef struct elem {
    int data;
    struct elem *left;
    struct elem *right;
} node;

node* search(node *tree, int val) {
    if(!tree) {
        return NULL;
    }

    if(val == tree->data) {
        return tree;
    } else if(val < tree->data) {
        return search(tree->left, val);
    } else if(val > tree->data) {
        return search(tree->right, val);
    }
}

void matrix_xor(int size, int A[size][size], int B[size][size], int C[size][size]) {
    int i, j, k, sum;

    for(i = 0; i < size; i++) {
        for(j = 0; j < size; j++) {
            sum = 0;
            for(k = 0; k < size; k++)
                sum = sum + (A[i][k] ^ B[k][j]);
            C[i][j] = sum;
        }
    }
}
```

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

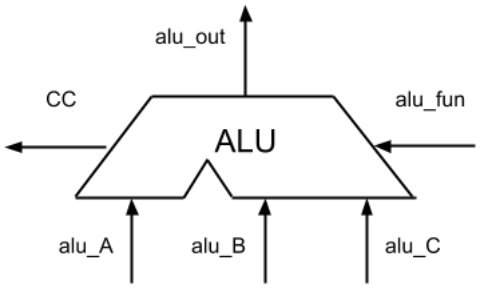


Figure 2: modified ALU.

- `lealX C(RA, RB)` will calculate `RA+RB*X+C` and store the value in `RB`. So, `leal0 C(RA,RB)` is similar to the x86 `leal C(RA),RB` and the other `lealX C(RA,RB)` are like `leal C(RA,RB,X),rB`.
- The `X` in the above instruction should be replaced by `0, 1, 2, 4` or `8`. So, you will implement 5 `leal` instructions in total for each processor.
- The `lealX` instructions should NOT modify the condition codes to follow the behavior of the `lea` instructions in x86.
- Table 1 shows the encoding for the Y86 `lealX` instructions.

Table 1: Y86 `lealX` instruction encoding

| | Byte 0 | Byte 1 | Byte 2 to Byte 5 |
|---|---|---|---|
| `leal0 C(RA, RB)` | `D F` | `RA RB` | `C` |
| `leal1 C(RA, RB)` | `D 0` | `RA RB` | `C` |
| `leal2 C(RA, RB)` | `D 1` | `RA RB` | `C` |
| `leal4 C(RA, RB)` | `D 2` | `RA RB` | `C` |
| `leal8 C(RA, RB)` | `D 3` | `RA RB` | `C` |

To add these instructions, you will modify the file `seq-full.hcl` in `sim/seq` directory and `pipe-full.hcl` in `sim/pipe`, which implement the version of SEQ and PIPE simulators respectively. You will also modify the `isa.c` file in `sim/misc` directory to allow the ALU to perform `lealX` instruction.

Some modification we made:

- We have modified the ALU such that it can take up to 3 inputs (aluA, aluB and aluC). This is useful for the ALU to support the new `lealX` instruction to do a multiply/shift as well as adding three terms in a single step. Figure 2 shows the modified ALU.
- We have modified `ifun` for some operations. The `ifun` for an `ADD` operation is now changed to `A`, `SUB` to `B`, `AND` to `C` and `XOR` to `D`. The `ifun` for `lealX` instruction is described in Table 1.

## 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. Similarly, once you have finished modifying the `pipe-full.hcl` file, you will need to build a new instance of the PIPE simulator (`psim`) and test it. The following instructions demonstrate how to test the SEQ and PIPE simulator.

- *Building a new simulator.* You can use the provided `Makefile`, in the `seq` directory for `seq-full.hcl`, 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.

  Similarly, You can use the provided `Makefile`, in the `pipe` directory for `pipe-full.hcl`, to build a new PIPE simulator:

  ```
  unix>  make VERSION=full
  ```

  This builds a version of `psim` that uses the control logic you specified in `pipe-full.hcl`. To save typing, you can assign `VERSION=full` in the Makefile.

- *Testing your solution using the benchmark programs.* You can automatically test your simulator on the Y86 benchmark programs in `../y86-code` on both the SEQ simulator and the PIPE simulator. To do this, you should run `make testssim` and `make testpsim` from the `y86-code` directory:

  ```
  unix>  make testssim
  unix>  make testpsim
  ```

  This will run `ssim` and `psim` 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 the `ptest` directory. To test your implementation of `lealX` instruction you should include `TFLAGS` argument with a `-l` flag. For example, to test your implementation of `lealX` in `seq-full.hcl`:

  ```
  unix>  make SIM=../seq/ssim TFLAGS=-l
  ```

  To test your implementation of `lealX` in `pipe-full.hcl`:

  ```
  unix>  make SIM=../pipe/psim TFLAGS=-l
  ```

For more information on the SEQ and PIPE simulators, refer to the handout *CS:APP2e Guide to Y86 Processor Simulators* (`simguide.pdf`).

# 6 Part C

Your task in Part C is to make the `matrix_xor.ys` run as fast as possible.

### Coding Rules

You are free to make any modifications you wish on your `matrix_xor.ys`, with the following constraints:

- Your `matrix_xor.ys` function must work for arbitrary matrix sizes which size is smaller than or equal to 8x8.
- Your `matrix_xor.ys` function must run correctly with YIS.

*Suggestion:* Make use of the `lealX` instruction if you have successfuly completed PART B.

### Building and Running Your Solution

In order to test your solution, you will need to build a driver program that calls your `matrix_xor` function. We have provided you with the `gen-driver.pl` program in the `pipe` directory that generates a driver program for arbitrary sized input matrix. In your `pipe` directory, typing:

```
unix>  make drivers
```

will construct the following two useful driver programs:

- `sdriver.yo`: A *small driver program* that tests a `matrix_xor` function on small matrices with 2x2 elements.
- `ldriver.yo`: A *large driver program* that tests a `matrix_xor` function on larger matrices with 8x8 elements each.

Each time you modify your `matrix_xor.ys` program, you can rebuild the driver programs by typing

```
unix>  make drivers
```

To test your solution in GUI mode on a small 2x2-element matrix, type

```
unix>  ./psim -g sdriver.yo
```

To test your solution on a larger 8x8-element matrix, type

```
unix>  ./psim -g ldriver.yo
```

# 7 Evaluation

The lab is worth 200 points: 40 points for Part A, 100 points for Part B, and 60 points for Part C.

## Part A

Part A is worth 40 points. 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 `search.ys` will be considered correct if the graders do not spot any errors in them.

The program `matrix_xor.ys` will be considered correct if the graders do not spot any errors in them, and the destination matrix contains the correct values.

## Part B

This part of the lab is worth 100 points:

- 10 points each for passing the tests for each `lealX` instruction in `seq-full.hcl`.
- 10 points each for passing the tests for each `lealX` instruction in `pipe-full.hcl`.

## Part C

This part of the Lab is worth 60 points:

- 60 points for performance. To receive credit here, your solution must be correct, as defined earlier.

  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 perform `matrix_xor` function on two matrices with $N^2$ elements, then the CPE is $C/(N^2)$. The PIPE simulator displays the total number of cycles required to complete the program.

  Since some cycles are used to set up the call to `matrix_xor` and to set up the loop within `matrix_xor`, you will find that you will get different values of the CPE for different matrix sizes. We will therefore evaluate the performance of your function by computing the average of the CPEs for matrices with size ranging from 1x1 to 8x8 elements. You can use the Perl script `benchmark.pl` in the `pipe` directory to run simulations of your `matrix_xor.ys` code over a range of block lengths and compute the average CPE. Simply run the command

  ```
  unix>  ./benchmark.pl
  ```

  to see what happens. Note that this Perl script does not check for the correctness of the answer. To check the correctness of the answer, you should use the `correctness.pl` script with a `-p` flag:

  ```
  unix>  ./correctness.pl -p
  ```

To get credits in this part, your average CPE must be less than 206.0. You should be able to achieve an average CPE of less than 186.0. If your CPE is $c$, then your score $S$ for this part will be:

$$S = \begin{cases} 0\,, & c > 206.0 \\ 3.0 \cdot (206.0 - c)\,, & 186.0 \leq c \leq 206.0 \\ 60\,, & c < 186.0 \end{cases}$$

By default, `benchmark.pl` and `correctness.pl` compile and test `matrix_xor.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 sets of files:
  - Part A: `search.ys` and `matrix_xor.ys`.
  - Part B: `seq-full.hcl`, `pipe-full.hcl` and `isa.c`.
  - Part C: `matrix_xor.ys`.
- Put your files for each part in a separate directory (`part_a`, `part_b`, `part_c`).
- Compress all these files into a single .zip file and submit it in the Moodle link for Architecture Lab.

# 9 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 object file.