# PKU-ICS
# Arch Lab: Optimizing the Performance of a Pipelined Processor

## 1 Introduction

In this lab, you will learn about the design and implementation of a pipelined Y86-64 processor, optimizing both it and a benchmark program to maximize performance. You are allowed to make any semantics-preserving transformation to the benchmark program, or to make enhancements to the pipelined processor, or both. When you have completed the lab, you will have a deep understanding of the interactions between code and hardware that affects 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 Y86-64 programs and become familiar with the Y86-64 tools. In Part B, you will first extend the SEQ simulator with new instructions, then explore the road of CPU piplining. These two parts will prepare you for Part C, the heart of the lab, where you will optimize the Y86-64 benchmark program and its architecture design.

## 2 Logistics

You will work on this lab alone. Any clarifications and revisions to the assignment will be posted on the course Web page.

## 3 Handout Instructions

1. You can do the lab on a Linux system.
2. Start by copying the file archlab-handout.tar to a directory in which you plan to do your work.
3. Then give the command: `tar xvf archlab-handout.tar`. This command extracts files to the same directory of archlab-handout.tar. You should check README to find out what these files are for.
4. Go to `archlab-project` directory and build the Y86-64 simulator tools with the help of `archlab-project/README.md`.

## 4 Environment Installation

This project is written in Rust, so you'd have your Rust toolchain installed. If you haven't, please execute the following command to install `rustup`. This installation requires network access. Therefore make sure to connect to the gateway via `clabcli connect` before installation.

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

You can verify the installation by executing the command `rustup`.

Now install the Rust toolchain by executing the following command (by the time of writing, the latest stable version is 1.90):

```
rustup install 1.90
rustup default 1.90
```

## 5 Build the Project

**Every time after applying changes to Rust source code (`*.rs`), you should rebuild the project to make up-to-date binaries**:

```
(cd archlab-project; cargo build)
```

After running this command, a folder `archlab-project/target` will be created to store the output binaries and other intermediate files. The output executables are

- Y86-64 Assembler: `target/debug/yas`
- Y86-64 Debugger: `target/debug/ydb`
- Y86-64 ISA Simulator: `target/debug/yis`
- Y86-64 Pipeline Simulator: `target/debug/ysim`
- Local Grader for Part A, Part B, and Part C: `target/debug/grader`

  You can learn more about their usage in `archlab-project/README.md`.

## 6 Part A

You will be working in directory `archlab-project/misc` in this part. 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 `examples.c`. Be sure to put your name and ID(e.g. 2300054321) in a comment at the beginning of each program. 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 Y86-64 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. Test your program using the following three-element list:

```
# Sample linked list
.align 8
ele1:
        .quad 0x00d
        .quad ele2
ele2:
        .quad 0x0e0
        .quad ele3
ele3:
        .quad 0xf00
        .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`. Test your program using the same three-element list you used for testing `sum.ys`.

### `bubble.ys`: Sort a block of ints(8 byte) in ascending order using bubble-sort

Write a program (`bubble.ys`) that sorts a block of ints(8 bytes) in-place, with the result in ascending order. That is, you may copy the numbers from one part of memory to another, but the answer should be in the same piece of memory.

Your program should consist of code that sets up a stack frame, invokes a function `bubble_sort`, and then halts. The function should be functionally equivalent to the C function `bubble_sort` Test your program using the following six-element source and destination blocks:

```
.align 8
Array:
        .quad 0xbca
        .quad 0xcba
        .quad 0xacb
        .quad 0xcab
        .quad 0xabc
        .quad 0xbac
```

## 7 Part B

You will be working in directory `archlab-project/sim/src/architectures/extra`.

### Extend the SEQ architecture with **`iopq`** instruction

Your first task in Part B is to extend the SEQ processor to support the following instructions:

- `iopq V, rB`: compute `rB op V` and store the result in register `rB`.

To add these instructions, you will modify the file `seq_full.rs`, which implements the version of SEQ described in the CS:APP3e textbook. The format of `iopq` looks like a combination of `irmovq` and `opq`. We've provide instruction code `IOPQ` for you. The function code of `iopq` is the same as that of `opq`.

After implementing this extension, you can write `iaddq`, `isubq`, `ixorq` in your Y86 assembly code.

### Road to a pipelined architecture

The textbook describes little details about how the SEQ (or SEQ+) architecture changes into the PIPE architecture. Therefore, in this part, you will be given 8 architectures in order:

- `pipe_s2`: An example of 2-stage pipeline, containing instruction fetch stage and all the rest part as decode stage.
- `pipe_s3a,pipe_s3b,pipe_s3c,pipe_s3d`: 3-stage pipelines, containing instruction fetch stage, decode stage and all the rest part as execute stage.
- `pipe_s4a,pipe_s4b,pipe_s4c`: 4-stage pipelines, containing instruction fetch stage, decode stage, execute stage and all the rest part as memory stage.

Starting from `pipe_s2`, which is modified from the SEQ+ architecture, each architecture makes some minor improvements based on the previous one, which effectively shows a possible evolution of CPU pipelining.

| HCL (textbook & previous labs) | HCL-rs (current lab) |
|---|---|
| `D_icode, E_icode, F_predPC, ...` | `D.icode, E.icode, F.pred_pc, ...` |
| `INOP, IHALT, IRRMOVQ, IIRMOVQ` | `NOP, HALT, CMOVQ, IRMOVQ` |
| `SADR, SINS, SBUB` | `Adr, Ins, Bub` |
| `word f_icode = [...];` | `u8 f_icode = [...];` |
| `word f_predPC = [...];` | `u64 f_pred_pc = [...];` |

Table 1: Differences between legacy HCL and HCL-rs

Among the above architectures, except `pipe_s2`, some expressions in the HCL descriptions are masked by placeholders. Your task is to read the HCL descriptions of each architecture and replace each placeholder by a correct expression. There are 3 types of placeholders: BOOL_PLACEHOLDER, U8_PLACEHOLDER, and U64_PLACEHOLDER. The prefix of a placeholder indicates the value type of its expression.

After completing all architectures, you should gain a deeper understanding of pipelining, which can help you overcome some challenges in part C.

Suggestion: Read the comments around each placeholder carefully.

### Notice

In order to provide a better lab experience with greater flexibility and modern development practices, we reimplement the HCL parser and simulator in Rust. With the help of the VSCode extension `rust-analyzer`, we can now enjoy syntax highlighting, auto-completion, type checking, and error reporting when writing HCL source code.

As a result, the hardware control language used in this lab (HCL-rs) is a bit different from previous one, whose specification can be found at `archlab-project/assets/hcl-rs.pdf`. To give a brief summary, the table 1 shows the major differences.

You should only pay attention to the HCL description in `sim_macros::hcl` block, and pipeline register definitions in `crate::defin_stages` block.

## 8 Part C

You will be working in directory `archlab-project/misc` and `archlab-project/sim/src/architectures/extra` in this part.

The `ncopy` function copies a `len`-element integer array `src` to a non-overlapping `dst`, returning a count of the number of positive integers contained in `src`.The C description of `ncopy` is in `misc/ncopy.c`.

Your task in Part C is to modify `archlab-project/misc/ncopy.ys` and `archlab-project/sim/src/architectures/extra/ncopy.rs` with the goal of making `ncopy` run as fast as possible.

- `archlab-project/misc/ncopy.ys` assembly file of `ncopy` function.

- `archlab-project/sim/src/architectures/extra/ncopy.rs` the description of CPU architecture that the `ncopy` function runs on.

You will be handing in these two files. Each file should begin with a header comment with the following information:

- Your name and ID.
- A high-level description of your code. In each case, describe how and why you modified your code.

### Coding Rules

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

- Your `ncopy` 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 `ncopy` function must run correctly with `yis`. By correctly, we mean that it must correctly copy the `src` block *and* return (in `%rax`) the correct number of positive integers.
- Size of assembled version of `ncopy` plus stack size is limited to 4Kb.(A little less than 4Kb in fact, you can check the grader code for an exact value) We will set the stack register and argument registers well before calling your `ncopy`.
- Your `ncopy.rs` implementation must pass the correctness tests for general y86-64 code.

Other than that, you are free to implement other instructions if you think that will help. You may make any semantics preserving transformations to the `ncopy` function, such as reordering instructions, replacing groups of instructions with single instructions, deleting some instructions, and adding other instructions. You may find it useful to read about loop unrolling in Section 5.8 of CS:APP3e.

## 9 Grade your solution

After finishing some parts of the lab, you should first rebuild the project and then execute the grader:

```
cd archlab-project
cargo build

# If you want to grade part A:
./target/debug/grader part-a

# If you want to grade part B:
```

```
./target/debug/grader part-b

# If you want to grade part C:
./target/debug/grader part-c
```

You may execute `./target/debug/grader -h` for grader usage.

## 10 Evaluation

The lab is worth 100 points: 15 points for Part A, 25 points for Part B, and 60 points for Part C. You can run the follow command to grade your implementation locally:

```
(cd archlab-project; cargo run --bin grader)
```

**The remote machine uses the same grader and all the checks(including validation test and length test...) are included.** Note that the score here does not include the credits for your descriptions.

### 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 `0xfed` in register `%rax`.

The program `bubble.ys` will be considered correct if the graders do not spot any errors in them, and the `bubble_sort` function sorts the 6 integers correctly in ascending order, with the results in the same 48 bytes beginning at address `Array`, and does not corrupt other memory locations.

### Part B

This part is worth 25 points, 4 points for the implementation of `seq_full.rs`, and 3 points for each of `pipe_s3a,pipe_s3b,pipe_s3c,pipe_s3d,pipe_s4a,pipe_s4b,pipe_s4c`.

For `seq_full.rs`, you need to pass the ISA checks extended with the `iopq` instruction.

For `pipe_s3a,pipe_s3b,pipe_s3c,pipe_s3d,pipe_s4a,pipe_s4b`, each of the architecture is required to pass ISA checks. Moreover, for each of them, the runtime status of each CPU cycle is compared with its corresponding ground truth architecture to verify the correctness of each placeholder expression. Since the ground truths are not provided in the student's handout, this check is performed on autolab server.

## Part C

This part of the Lab is worth 60 points for performance. You will not receive any credit if either your code for `ncopy.ys` or your modified architecture `ncopy.rs` fails.

We will evaluate the performance based on **cpe** (cycles per element) and **ac** (architecture cost) of your implementation.

- **cpe**: if the simulated code requires $C$ cycles to copy a block of $N$ elements, then the CPE is $C/N$. 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.

  Simply run the command

  ```
  (cd archlab-project; cargo run --bin grader -- part-c)
  ```

  to see what happens. For example, the baseline version of the `ncopy` function and architecture has CPE values ranging between 23.00 and 12.04, with an average of 12.82.

- **ac**: the length of the critical path of the ncopy architecture. Formally, the critical path of a CPU architecture is the longest path of combinational logic between clocked elements (like flip-flops). The length of the critical path can be used to mesure the CPU's clock frequency, which in turn can be used to estimate the architecture performance.

  In this lab, the length of the critical path is simplified as: 1 plus the maximum number of hardware devices (units) that line up in a path of the architecture. For example, `seq_std` has a critical path of length 8, and `pipe_std` has a critical path of length 4.

  You can execute

  ```
  (cd archlab-project; cargo run --bin ysim -- -A [arch_name] -I)
  ```

  to inspect the length of the critical path and the devices execution order of an architecture. This command will also generate an HTML file that visualizes the dependency graph of the architecture.

Let $c = cpe + 2 \times ac$. Your score *S* for Part C will be:

$$S = \begin{cases} 0\,, & c > 19.0 \\ 19 \cdot (19.0 - c)\,, & 16.0 < c \leq 19.0 \\ 57\,, & 15.0 < c \leq 16.0 \\ 60\,, & c \leq 15.0 \end{cases}$$

## 11 Handin Instructions

- You will be handing in three sets of files:
  - Part A:

    ```
    archlab-project/misc/bubble.ys
    archlab-project/misc/sum.ys
    archlab-project/misc/rsum.ys
    ```

  - Part B:

    ```
    archlab-project/sim/src/architectures/extra/seq_full.rs
    archlab-project/sim/src/architectures/extra/pipe_s3a.rs
    archlab-project/sim/src/architectures/extra/pipe_s3b.rs
    archlab-project/sim/src/architectures/extra/pipe_s3c.rs
    archlab-project/sim/src/architectures/extra/pipe_s3d.rs
    archlab-project/sim/src/architectures/extra/pipe_s4a.rs
    archlab-project/sim/src/architectures/extra/pipe_s4b.rs
    archlab-project/sim/src/architectures/extra/pipe_s4c.rs
    ```

  - Part C:

    ```
    archlab-project/misc/ncopy.ys
    archlab-project/sim/src/architectures/extra/ncopy.rs
    ```

- Make sure you have included your name and ID in a comment at the top of each of your handin files.
- To create your handin files for the lab, go to your `archlab-handout`. Run `make handin` to create archlab-handin.tar. Upload this tar to autolab for grading.

## 12 Hints

- `ysim -A [arch] -I` can generate an HTML file that visualizes the architecture computational dependency graph. `-v` can be used to print detailed information of each cycle. `--max-cpu-cycle` can be used to limit the number of CPU cycles.
- All Rust source files under `archlab-project/sim/src/architectures/extra`, except `mod.rs`, are considered as CPU architectures. If you want to create a new architecture, just create a new file there. `ydb` can debug your custom architecture via `--arch` option.
- `yis` simulates your program w.r.t. the Y86 ISA specification. Its output can be seen as ground truth.
- In part B, you can rely on editor features to display the differences between two files. For `vim` users you can use `vimdiff`. For VSCode users you can first open one of the source files, then goto "Help > Show All Commands" and type "Compare Active File With...", and select another file for comparison (at this point you're able to edit both files concurrently).

- In part C, the default HCL description `ncopy.rs` is simply a copy of `seq_std.rs`. You may want to replace it with another pipelined architecture, and then apply some modifications on it.