Middle East Technical University Department of Computer Engineering

CENG 331

Computer Organization

Fall ’2023-2024

THE3: Architecture Lab

Optimizing the Performance of a Pipelined Processor

Due date: Dec 24 2023, Sunday, 23:55

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 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 Y86-64 programs and become familiar with the Y86-64 tools. In Part B, you will extend the SEQ simulator with a new instruction. These two parts will prepare you for Part C, the heart of the lab, where you will optimize the Y86-64 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 ODTUClass.

3 Handout Instructions

1. Start by copying the file `archlab-handout.tar` to a directory in which you plan to do your work.
2. Then give the command: `tar xvf archlab-handout.tar`. This will cause the following directories files to be unpacked into the directory: `README`, `sim`, `archlab.pdf`, and `simguide.pdf`.
3. Finally, change to the `sim` directory and build the Y86-64 tools:

```
unix>  cd sim
unix>  make clean && make
```

Note that this should work directly on the ineks, but you'll need to do some extra work if you want to compile the lab on your own system. Check the final section, **Installation & Usage Hints**.

4 Brand New Arithmetic Instructions

For this assignment, your Y86 processors come prequipped with instructions (you do not need to modify `seq-full.hcl` or `pipe-full.hcl` to use them):

- `mllq %rA, %rB` multiplies the contents of two registers and stores the result in `%rB`. The upper bytes are truncated (this has bad implications, see Known Issues).
- `dvvq %rA, %rB` divides `%rB` by `%rA` and stores the result in `%rB`.
- `modq %rA, %rB` takes the mod of `%rB` by `%rA` and stores the result in `%rB`.

5 Part A

You will be working in directory `sim/misc` in this part. To build the assembler and the simulator themselves, run `make` within this directory.

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 `sample_files/high_level_implementations.c`. Be sure to put your name and ID in a comment at the beginning of each program. You can test your programs by first assembling them with the program YAS:

```
yas <path_to_your_code.ys>
```

and then running them with the instruction set simulator YIS:

```
yis <path_to_your_object_file.yo> [number_of_steps]
```

`yis` takes an optional number that is the number of instructions it should execute, but this is often not so useful for debugging. A better alternative is using the gui mode of `ssim` or `psim`.

In all of your Y86-64 functions, you should follow the x86-64 conventions for passing function arguments, using registers, and using the stack (The first argument is in `%rdi`, second in `%rsi`, third in `%rdx`, fourth in `%rcx`, fifth in `%r8`. Stack pointer is held in `%rsp`. Return value is stored in `%rax`).

Iterative Binary Search

Implement `binary_search_it()` specified in `sample_files/high_level_implementations.c` in Y86 assembly using the template for binary search.

Specifications and Hints

- Your input array will be sorted in ascending order, filled with non-negative 8 byte integers that appear only once in the array (they are still of type `long` (a signed type) in C, we are doing this to avoid the menace of the `mllq` quirks.
- The array length will be in the range $[0, 30]$ (inclusive).
- Each element will be in the range $[1, 1000]$ (inclusive).
- Note that we add 1 to the final index! For example, if n is the first element in the array, we return 1, not 0. If it didn't exist, we would have returned -1.
- Your implementation will be pasted to different instances of assemblies with different arrays and query values (our own tiny linker). To do a verification of your implementation, you can use the python script under `verifiers` directory:

  ```
  python3 binary_search_verifier.py yis <path_to_your_implementation.ys>
  ```

  This script will:

  – paste your implementation region to assembly files generated on the fly, containing test cases, putting them in a new directory,
  – compile them using `yas`,
  – run it with `yis` (or `ssim`,`psim` if you replace the second argument), check if the program correctly terminates and returns the correct index in `%rax`.
  – You are free to tinker with it to debug/test other scenarios. You can run the problematic test case object files individually by hand with `yas`, `ssim` or `psim`.
- The random seed, number of test cases, and certain offsets within the assembly (stack pointer location, array locations etc.) will be changed during grading, so do not try hard coding those parts.
- Your instruction code or PC should not appear with the addresses above 0x1000, with .pos directives or otherwise. This part is where the data and the stack lays in the test cases.
- The stack size should not exceed 768 bytes.

Recursive Binary Search

Implement the same algorithm but recursive this time, i.e. `binary_search_rec()` in Y86 assembly using the template for binary search. Same specifications apply with the iterative case, **however your program execution should call the label `binary_search` at least once within this subroutine.**

Merge

Implement `merge()` in Y86 assembly using the template for merge.

Specifications

- The input arrays will be sorted in ascending order, filled with non-negative 8 byte integers that appear only once in the array.
- The array lengths will be in the range $[0, 30]$ (inclusive).
- Each element will be in the range $[1, 1000]$ (inclusive).
- This task has a verifier that works exactly the same as the binary search verifier. To run it, simply

  ```
  python3 merge_verifier.py yis <path_to_your_implementation.ys>
  ```
- The random seed, number of test cases, and certain offsets within the assembly (stack pointer location, array locations etc.) will be changed during grading, so do not try hard coding those parts.
- Your instruction code or PC should not appear with the addresses above 0x1000, with .pos directives or otherwise. This part is where the data and the stack lays in the test cases.
- The stack size should not exceed 768 bytes.

6 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 a new instruction, `jtab`, a ten-byte instruction having the form `jtab C, %rB`. This instruction corresponds to the indirect jump instruction in x86 –`jmp*`. Upon execution of this instruction, the program counter should be updated with the value stored in `C(%rB)` (not `C(%rB)` itself!). The instruction has the same signature as `irmovq` instruction, which takes a single register byte for register B and an 8 byte constant value. The instruction should emit an `ADR` exception if the address `C(%rB)` is invalid. It should also emit an `INS` exception if the stored address within `C(%rB)` is invalid. The instruction does not set any condition codes.

To add this instruction, you will modify the file `seq-full.hcl`, which implements the version of SEQ described in the CS:APP3e 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 your surname, name and ID. Add also a paragraph that summarizes your modifications with reasoning.

Testing SEQ

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.* Within the `seq` directory, you can use `make` 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.
- *Testing your solution on a simple Y86-64 program.* For your initial testing, we recommend running simple test programs such as `jtabtest*.yo` (source code is in `sample_files`) in TTY mode, comparing the results against the ISA simulation:

```
unix>  ./ssim -t <path_to_jtab_test_yo>
```

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

```
unix>  ./ssim -g <path_to_jtab_test_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`:

  ```
  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.

- *Performing regression tests.* Once you can execute the benchmark programs correctly, then you should run the extensive set of regression tests in `../ptest`:

  ```
  unix>  (cd ../ptest && make SIM=../seq/ssim)
  ```

- **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. For `jtab` itself, either use `jtabtest*.yo` files or write your own**.

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

7 Part C

You will be working in directory `sim/pipe` in this part. Your task is to implement `switch8()` specified in `sample_files/high_level_implementations.c` as fast as possible, by starting from `switch8_slow.ys`.

The file `pipe-full.hcl` contains a copy of the HCL code for PIPE, along with constant declarations for instruction codes.

You will be handing in two files: `pipe-full.hcl` and `switch8.ys`. Each file should begin with a header comment with the following information:

- Your name and ID.
- A high-level description of your code. For `switch8.ys`, describe how and why you modified your code. A step by step approach is recommended for clarity, e.g. - I did X reducing my CPE from A to B, - I did Y reducing my CPE from B to C etc. For `pipe-full.hcl`, describe how and why you modified the control logic and how this was helpful in speeding up your code. Note that you can also choose to not modify `pipe-full.hcl` at all and keep your optimizations restricted to the code.

Specifications

- The input array to be processed is composed of non-negative `long` integers in the range $[1, 1000]$. Each branch of the switch takes almost equal number of elements in the test cases.
- The input array is of length in the range $[1, 64]$. In the test cases, maximum length arrays are the overwhelming majority.

- You can test your solution on yis (regardless of whether you have implemented PIPE or not) with:

  ```
  python3 switch8_verifier.py yis <path_to_switch8.ys>
  ```

  With this script (or `yis` itself if you are testing by hand) you can even test solutions that use `jtab` instruction without implementing the instruction itself. You can also use this script to test your part B solution again by replacing `yis` with `ssim`. This will do an ISA check for your SEQ implementation by comparing it with the ground truth execution of `yis` (if your do it by hand, it corresponds to -t flag in `ssim`).

- The random seed, number of test cases, and certain offsets within the assembly (stack pointer location, array locations etc.) will be changed during grading, so do not try hard coding those parts.
- Your instruction code or PC should not appear with the addresses above 0x1000, with .pos directives or otherwise. This part is where the data and the stack lays in the test cases.
- The stack size should not exceed 768 bytes.

You are free to make any modifications you to PIPE, Your `pipe-full.hcl` implementation must pass the regression tests in `../y86-code`. To execute the tests, within `../y86-code`:

```
unix> make testpsim
```

You may make any semantics preserving transformations, 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. You are allowed to add constant data (such as arrays, as you did in part A to your program using the directives `.align`, `.quad` and `.pos` (this should not affect regions with adresses larger than 0x1000 however).

Testing PIPE

- Within `pipe` directory, each time you modify your `pipe-full.hcl` file, you can rebuild the simulator by typing

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

- You are free to make any modifications you to PIPE, Your `pipe-full.hcl` implementation must pass the regression tests in `../y86-code`. To execute the tests, within `../y86-code`:

  ```
  unix> make testpsim
  ```

- Once you can execute the `../y86-code` programs correctly, then you should check it with the regression tests in `../ptest`:

  ```
  unix>  (cd ../ptest && make SIM=../pipe/psim)
  ```

- Finally, for `jtab`, running `switch8_verifier.py` with `psim` will provide CPE: *cycles per element*. This measures how many cycles are spent average on an element to process it. This metric will ultimately be used to grade your optimization.

8 Evaluation

The lab is worth 165 points: 45 points for Part A, 30 points for Part B, and 90 points for Part C.

Since this homework is more conventional than the Bomb and Attack Labs, your handins will be checked for plagiarism, as per usual. Please remember that **we have a zero tolerance policy for cheating**. This includes any work that is not your own, including using sources from the internet.

Part A

Part A is worth 45 points, 15 points for each Y86-64 solution program. Each solution program will be evaluated for correctness with a modified version of the verifiers.

Part B

This part of the lab is worth 30 points:

- 5 points for explanations and comments regarding your hardware implementation.
- 2.5 points for passing the benchmark regression tests in `y86-code`, to verify that your simulator still correctly executes the benchmark suite.
- 2.5 points for passing the regression tests in `ptest`.
- 5 points for correct execution of example files `jtabtest*.yo`.
- 15 points for correct execution of answer key `switch8` implementation with `jtab`. (you do not need to provide it)

Part C

This part of the Lab is worth 90 points:

- 5 points for explanations and comments regarding your hardware implementation and asm optimization.
- 15 points in total for each correctly handled test case. **To be eligible for the subsequent items, you need to handle every test case correctly, and your PIPE implementation should pass all regression tests (both under `y86-code` and `ptest`)**.
- 5 points for correct execution of example files `jtabtest*.yo`.
- 15 points for correct execution of answer key `switch8` implementation with `jtab`. (you do not need to provide it)
- 50 points for performance. We will express the performance of your function in average CPE with

  ```
  unix> switch8_verifier.py psim <path_to_your_switch8>
  ```

  The baseline version of the `switch8` function running on the standard PIPE simulator has average CPE of 45.61.

  If your average CPE is c, then your score S for this portion of the lab will be:

$$S = \begin{cases} 0\,, & c > 36.5 \\ 6.25 \cdot (36.5 - c)\,, & 28.5 \leq c \leq 36.5 \\ 50\,, & c < 28 \end{cases}$$

```
    .pos 0
    # initial code for setting
    # up the stack and calling main or your function
    # and stopping after your function returns

# the example data, starting at
# byte 512 to be far enough from
# your initial code to not have problems
.pos 0x200
    # .. data ..
    # .. data ..
    # .. data ..

main:
    # Optionally, you can have a main function
    # setting up the arguments to your function
    # and calling it, but it’s optional. Feel
    # free to call func directly from the initial code.

func:
    # code for your function...
    # .. code ..
    # .. code ..
    # .. code ..
    # .. code ..

# stack starting at byte 2048,
# far away from the code, your code
# should not be long enough to get here anyway!
.pos 0x800
stack:
```

Figure 1: **An example layout for the functions in part A.** Check `y86-code/asum.ys` for an example.

9 Tips and Tricks

Part A

- Be careful with the placement of the absolutely positioned data, and make sure to place the stack far enough from your code since it grows downwards (towards zero). The simulator will not think twice about overwriting your code if the stack grows too large, which might be hard to debug. An example layout that should work is shown in Figure 1.

  You can check examples under the `y86-code` directory (such as `asum.ys`) if you want to see how the initial set-up code is written. Remember that having a `main` function is optional.

- Examine the value of `%rax` and the `Changes to memory` section from the output of the ISA simulator YIS to make sure that your functions work.

- In the CS:APP3e book, Figure 4.1 (around page 383) shows the Y86-64 registers while Figure 4.2 (around page 385) shows the Y86-64 instruction set.

Part B

- You do not have full control over the circuit design of the processor, instead, you can modify the existing control logic, which makes your job simpler. This part is much easier than it seems initially!
- Figure 4.23 (around page 427) in the CS:APP3e textbook illustrates the design of SEQ, which might be helpful.

Part C

- There is a very obvious algorithm to eliminate the ugly linear search as if-else statements. If you do not want to attempt implementing `jtab` for a single jump, you can opt for this algorithm for partial grade. The author of this text has used that algorithm for defining a CPE bound for partial grades.
- Even though your code needs to work for all block sizes, the benchmark is the average CPE for block sizes from 1 to 64 only. Larger sizes are the majority in the CPE calculation.
- Remember that PIPE does not re-order instructions. You have to consider possible hazards that may delay the pipeline using your own knowledge. Think hard about the program and do your best to write correct code that is as fast as possible.
- `pipe-full.hcl` may seem impenetrable at first. It is not! There are different modifications you can perform that could help with performance, depending on the structure of your program. As a bonus, tinkering with `pipe-full.hcl` will help you understand PIPE much better. This knowledge may be useful in the written exams.
- For handling various hazards, think of similar instructions to `jtab` and observe how they are handled. It might be that some hazards have already been passively taken care of for other instructions and therefore `jtab` too.
- Figure 4.52 (around page 468) in the CS:APP3e textbook illustrates the design of PIPE, which will be helpful.

10 Handin Instructions

- You will submit your solutions as a single compressed archive file named `eXXXXXXX.tar.gz` to ODTU-Class, where `XXXXXXX` is your 7-digit student ID. Please name your file correctly. Remember that you can create .tar.gz (gzipped tarball) files as follows:

```
unix>  tar -czf eXXXXXXX.tar.gz <files>
```

- Your archive should contain the following files:
  - Part A: `binary_search_it.ys`, `binary_search_rec.ys`, and `merge.ys`.
  - Part B: `seq-full.hcl`.
  - Part C: `switch8.ys` and `pipe-full.hcl`.

  These files should all be directly under the archive; your archive should not contain any directories.
- Make sure you have included your name and ID in a comment at the top of each of your handin files.

11 Known Issues

- `yas` might skip the very last instruction/directive you put in a file if it is at the last line. Make sure your .ys files have an empty line at the end.
- Dividing or taking mod with 0 with `dvvq` or `mllq` causes a figurative halt and catch fire in the simulators by throwing a floating-point exception. You will not need to do these operations in any part of your assignment.
- Overflow with `mllq` instruction has inconsistent behaviour with the signs of numbers due to the nature of multiplying negative numbers always resulting in overflow in the unsigned realm. Always work with positive numbers of at most 4 bytes with this instruction. You will not need to do otherwise for this assignment.
- Unfortunately `psim` or `ssim` do not run in gui mode when executed outside from their directories. You need to relocate within `pipe` or `sim`.
- 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.
- Jumping to an invalid instruction address with `jtab` *might* cause segmentation faults in GUI mode.

12 Installation & Usage Hints

- For your local Ubuntu 2022 system, you will need to install `bison`,`flex` packages through `apt`.
- In order to compile the simulator with GUI mode enabled, Tcl/Tk libraries are necessary. The `TKLIBS` and `TKINC` variables in the `Makefile`s are configured for 64-bit Linux and Tcl/Tk8.6, with the ineks in mind. Thus:
  - If you want to compile without GUI support, comment the `GUIMODE`, `TKLIBS` and `TKINC` variables out in the `Makefiles` under `sim`, `sim/seq` and `sim/pipe`.
  - If you have a 64-bit Linux system running Ubuntu, the following package installation commands should set you up:

```
ubuntu>  sudo apt update
ubuntu>  sudo apt install tcl-dev tk-dev
```

- 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.