Middle East Technical University Department of Computer Engineering

CENG 331

Computer Organization

Fall ’2022-2023

THE3: Architecture Lab

Optimizing the Performance of a Pipelined Processor

Due date: 24 December 2022, Saturday, 23:59

Important Note: We’re starting with a one-week extended deadline to give everyone time to properly digest the homework and maybe even work on bonuses. Please start early and give yourself time to learn and think!
There will be no other deadline extensions for this homework.

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 (protected) directory 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`, `sim.tar`, `archlab.pdf`, and `simguide.pdf`.
3. Next, give the command `tar xvf sim.tar`. 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:

   ```
   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 Part A

You will be working in directory `sim/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 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.

`kth_ll.ys`: Find the kth smallest value in an ordered linked list

Write a Y86-64 program `kth_ll.ys` to iteratively find the kth smallest value in an ordered (ascending) 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 (`kth_ll`) that is functionally equivalent to the C `kth_ll` function in Figure 1. Test your program using the following 7-element linked list:

```
# absolutely positioned here to debug addresses more easily.
.pos 0x200
ele0:
    .quad 0x0005
    .quad ele1
ele1:
    .quad 0x000c
    .quad ele2
ele2:
    .quad 0x0010
    .quad ele3
ele3:
    .quad 0x001a
    .quad ele4
ele4:
    .quad 0x0030
    .quad ele5
```

```
ele5:
    .quad 0x0045
    .quad ele6
ele6:
    .quad 0x005e
    .quad 0 # Remember that 0 is null.
```

`kth_bst.ys`: Recursively find the kth smallest value in a binary search tree

Write a Y86-64 program `kth_bst.ys` that recursively finds the kth smallest value of a binary search tree. In other words, you need to use the in-order tree traversal to find kth element. This code will have to use recursion, as shown with the C function `kth_bst` in Figure 1. Test your program using the following 11-element binary search tree.

```
# Absolutely positioned here to debug addresses more easily.
.pos 0x200
root:
    .quad 17
    .quad node6
    .quad node24
node6:
    .quad 6
    .quad node4
    .quad node11
node4:
    .quad 4
    .quad node3
    .quad node5
node3:
    .quad 3
    .quad 0
    .quad 0
node5:
    .quad 5
    .quad 0 # Remember that 0 is null.
    .quad 0
node11:
    .quad 11
    .quad node8
    .quad 0
node8:
    .quad 8
    .quad 0
    .quad 0
node24:
    .quad 24
    .quad node19
    .quad node40
node19:
    .quad 19
    .quad 0
    .quad 0
node40:
    .quad 40
    .quad 0
    .quad node52
node52:
    .quad 52
    .quad 0
    .quad 0

k:
    .quad 8
```

`insert_ll.ys`: Insert a new element to an ordered linked list

Write a program `insert_ll.ys` that inserts a new value to an ordered linked list, and returns the pointer to this new linked list. Note that the function takes an array with size of 2 and the new value to be inserted, as arguments. The new value can be the smallest value in the new linked list.

Your program should consist of code that sets up a stack frame, invokes a function `insert_ll`, and then halts. The function should be functionally equivalent to the C function `insert_ll` shown in Figure 1.
Test your program using the same linked list as in `kth_ll`, and insert a new element by storing necessary values in the following empty array having just 2 slots:

```
# An array with size of 2 to put new element in:
# Make sure your code works correctly. Do not
# fill beyond the bounds of the array. You should
# see the new value and the pointer to the next
# element.
.pos 0x400
array:
    .quad 0
    .quad 0
```

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 a new instruction, `isubq`, a ten-byte instruction having the form `isubq V, rB`. It will simply subtract the value in register `rB` from the constant value `V`, and store the result in register `rB`. If the constant value `V` is 0, `isubq` instruction will store the negated value back in register `rB`. The main takeaway is that you should be careful with the ordering of the registers. It should also *set condition codes*. The encoding of `isubq` is shown in Figure 2.

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 the following information:

```c
struct Node {
    long data;
    struct Node *left;
    struct Node *right;
};

struct list {
    long value;
    struct list *next;
};

long kth_ll(const struct list *head, long k)
{
    long kth = 0;
    while (head) {
        k--;
        if(k == 0) {
            kth = head->value;
            break;
        }
        head = head->next;
    }
    return kth;
}

long kth_bst(struct Node* root, long *k)
{
        if (root) {
            long candidate = kth_bst(root->left, k);
            if (k > 0) {
                -- *k;
                if (*k == 0)
                    return root->data;
                return kth_bst(root->right, k);
            }
            else {
                return candidate;
            }
        }
    return -1;
}

struct list * insert_ll(struct list *head, long *array, long value)
{
    *array = value;
    if (!head || head->value > value) {
        *(array+1) = head;
       return array;
    }
    else {
        struct list *temp = head;
        while(temp->next && temp->next->value < value )
            temp = temp->next;
        *(array+1) = temp->next;
        temp->next = array;
        return head;
    }
}
```

Figure 1: **C versions of the Y86-64 solution functions.** See `sim/misc/examples.c`

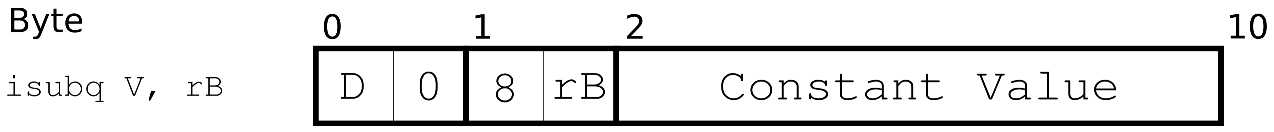

Figure 2: **Encoding of the isubq instruction**

- Your name and ID.
- A description of the computations required for the `isubq` instruction. Use the descriptions of `irmovq` and `OPq` in Figure 4.18 in the CS:APP3e 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 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 `isubqmany.yo` (testing `isubq`) in TTY mode, comparing the results against the ISA simulation:

  ```
  unix>  ./ssim -t ../y86-code/isubqmany.yo
  ```

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

  ```
  unix>  ./ssim -g ../y86-code/isubqmany.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. 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 `isubq`:

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

  To test your implementation of `isubq`:

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

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

```
/*
 * abscopy - copy the absolute values of elements contained in src to dst array,
 * return the sum of copied (absolute) values.
 */
word_t abscopy(word_t *src, word_t *dst, word_t n)
{
    word_t sum = 0;
    word_t val;
    word_t absval;

    while (n > 0) {
        val = *src++;
        absval = val > 0 ? val : -val;
        sum += absval;
        *dst++ = absval;
        n--;
    }
    return sum;
}
```

Figure 3: **C version of the `abscopy` function.** See `sim/pipe/abscopy.c`.

6 Part C

You will be working in directory `sim/pipe` in this part.

The `abscopy` function in Figure 3 copies the absolute values of `n`-element integer array `src` to a non-overlapping `dst` array, returning the sum of copied (absolute) values. Figure 4 shows the baseline Y86-64 version of `abscopy`. The file `pipe-full.hcl` contains a copy of the HCL code for PIPE, along with constant declarations for instruction codes.

Your task in Part C is to modify `abscopy.ys` and `pipe-full.hcl` with the goal of making `abscopy.ys` run as fast as possible.

You will be handing in two files: `pipe-full.hcl` and `abscopy.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 `abscopy.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.

Coding Rules

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

- Your `abscopy.ys` function must work for arbitrary array sizes. You might be tempted to hardwire your

```
#####################################################################
# abscopy.ys - copy the absolute values of a src block of n words to dst.
# Return the sum of copied (absolute) values.
#
# Include your name and ID here.
# Describe how and why you modified the baseline code.
#####################################################################
# Do not modify this portion
# Function prologue.
# %rdi = src, %rsi = dst, %rdx = n
abscopy:
#####################################################################
# You can modify this portion
        # Loop header
        xorq %rax,%rax          # sum = 0;
        andq %rdx,%rdx          # n <= 0?
        jle Done                # if so, goto Done:

Loop:
        mrmovq (%rdi), %r10     # read val from src...
        andq %r10, %r10         # val >= 0?
        jge Positive            # if so, skip negating
        rrmovq %r10, %r9        # temporary move
        xorq %r10, %r10         # zero r10
        subq %r9, %r10          # negation achieved!
Positive:
        addq %r10, %rax         # sum += absval
        rmmovq %r10, (%rsi)     # ...and store it to dst
        irmovq $1, %r10
        subq %r10, %rdx         # n--
        irmovq $8, %r10
        addq %r10, %rsi         # dst++
        addq %r10, %rdi         # src++
        andq %rdx,%rdx          # n > 0?
        jg Loop                 # if so, goto Loop:
#####################################################################
# Do not modify the following section of code
# Function epilogue.
Done:
        ret
#####################################################################
# Keep the following label at the end of your function
End:
```

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

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 `abscopy.ys` function must run correctly with YIS. By correctly, we mean that it must correctly copy the absolute values of first `n` elements of `src` list *and* return (in `%rax`) the correct sum of absolute values. The linked list `src` will contain more than `n` elements.
- The assembled version of your `abscopy` file must not be more than 1000 bytes long. You can check the length of any program with the `abscopy` function embedded using the provided script `check-len.pl`:

  ```
  unix>  ./check-len.pl < abscopy.yo
  ```

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

Other than that, you are free to implement the `isubq` instruction and apply changes to the control logic if you think that will help as long your `pipe-full.hcl` implementation passes the regression tests (i.e. the base Y86-64 instruction set remains functional). Once again, you can also choose to not modify `pipe-full.hcl` at all with no grade penalty.

You may make any semantics preserving transformations to the `abscopy.ys` 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. 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`.

Building and Running Your Solution

In order to test your solution, you will need to build a driver program that calls your `abscopy` 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:

- `sdriver.yo`: A *small driver program* that tests an `abscopy` function on small arrays with 4 elements. If your solution is correct, then this program will halt with a value of 10 (`0xa`) in register `%rax` after copying the `src` array taking the absolute values.
- `ldriver.yo`: A *large driver program* that tests an `abscopy` function on larger arrays with 63 elements. If your solution is correct, then this program will halt with a value of 2016 (`0x7e0`) in register `%rax` after copying the `src` array taking the absolute values.

Each time you modify your `abscopy.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 VERSION=full
```

If you want to rebuild the simulator and the driver programs, type

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

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 `abscopy.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 `abscopy.ys` function works properly with YIS:

  ```
  unix>  make drivers
  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 0 up to some limit (default 65), plus some larger sizes. It simulates them (by default with YIS), and checks the results. It generates a report showing the status for each block length:

  ```
  unix>  ./correctness.pl
  ```

  This script generates test programs where the result count varies randomly from one run to another, and so it provides a more stringent test than the standard drivers.

  If you get incorrect results for some length K, you can generate a driver file for that length that includes checking code, and where the result varies randomly:

  ```
  unix>  ./gen-driver.pl -f abscopy.ys -n K -rc > driver.ys
  unix>  make driver.yo
  unix>  ../misc/yis driver.yo
  ```

  The program will end with register `%rax` having the following value:

  `0xaaaa` : All tests pass.

  `0xbbbb` : Incorrect sum.

  `0xcccc` : Function abscopy is more than 1000 bytes long.

  `0xdddd` : Some of the source data was not copied to its destination.

  `0xeeee` : Some word just before or just after the destination region was corrupted.

- *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`:

  ```
  unix>  (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 `isubq` instruction, then to test `isubq` along with all the standard instructions:

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

- *Testing your code on a range of block lengths with the pipeline simulator.* Finally, you can run the same code tests on the pipeline simulator that you did earlier with the ISA simulator

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

7 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, including proper handling of the stack and registers, as well as functional equivalence with the example C functions in `examples.c`.

The program `kth_ll.ys` will be considered correct if the graders do not spot any errors in them, memory is not corrupted and the function returns `0x1a` in register `%rax` when it is called with $k = 4$.

Similarly, `kth_bst.ys` will be considered correct if the graders do not spot any errors in them, memory is not corrupted and the function returns `0x13` in register `%rax`. Remember the pointer k is given with the binary search tree in Part A and it stores the value 8.

Finally, `insert_ll.ys` will be considered correct if the graders do not spot any errors in them, and the function returns `0x200` in `%rax`, correctly inserts the new value 56 to the linked list, and does not corrupt other memory locations or go over the bounds of the array. And the necessary changes to memory are as follows:

```
0x0248: 0x0000000000000250       0x0000000000000400
0x0400: 0x0000000000000000       0x0000000000000038
0x0408: 0x0000000000000000       0x0000000000000250
```

Part B

This part of the lab is worth 30 points:

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

Part C

This part of the Lab is worth 90 points: **You will not receive any credit if either your code for `abscopy.ys` or your modified simulator fails any of the tests described earlier.**

- 15 points each for your descriptions in the headers of `abscopy.ys` and `pipe-full.hcl` and the quality of these implementations. To be extra clear, again, you do not have to modify `pipe-full.hcl` to get a full grade, your `abscopy.ys` will be graded out of 30 if you do not. However, in case you do modify it explanations must be present.
- 60 points for performance. To receive credit here, your solution must be correct, as defined earlier. That is, `abscopy` runs correctly with YIS (unless you modify instructions, in which case running on `psim` is enough) , and `pipe-full.hcl` passes all tests in `y86-code` and `ptest` (remember that `isubq` will not be tested during grading).

  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 displays the total number of cycles required to complete the program. The baseline version of the `abscopy` function running on the standard PIPE simulator with a large 64-element array requires 1010 cycles to copy 64 elements, for a CPE of $1010/64 = 15.78$.

  Since some cycles are used to set up the call to `abscopy` and to set up the loop within `abscopy`, 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 `abscopy.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 `abscopy` function has CPE values ranging between 36.00 and 15.78, with an average of 16.94. Note that this Perl script does not check for the correctness of the answer. Use the script `correctness.pl` for this.

  For 60 points(Speed-up): The 3 student with the lowest amount of CPEs (highest speedup) will receive 60 points. Rest of the grades will be scaled accordingly, based on their standing in between highest CPE score of the top 3 student and base line CPE count (16.94).

  By default, `benchmark.pl` and `correctness.pl` compile and test `abscopy.ys`. Use the `-f` argument to specify a different file name. The `-h` flag gives a complete list of the command line arguments.

8 Tips and Tricks

Part A

- The `examples.c` file shown in the PDF is stripped of all comments to fit in the PDF. Check the `examples.c` file under `misc` to see the real, commented version, if you want to understand what is going on.
- You do not necessarily have to think about the algorithms, simply reproducing the C versions of the given functions using Y86-64 is enough. Of course, you are free to write your own if you want to challenge yourself. This is fine as long as your functions behave in the expected way, as explained in the evaluation section.

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

- 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 5.

  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

- 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!
- 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. However, you can still choose not to modify it at all with no extra penalty.
- You cannot add new instructions to the ISA. However, since `isubq` will not be tested, you can change `pipe-full.hcl` to make `isubq` do something else entirely, if you have an idea that would help performance. Obviously the execution of your program will not match the ISA simulator YIS's execution in this case for programs containing `isubq`, and you should perform the regression tests without the `-i` flag. But this is fine since they will not be tested during grading. Make sure to always explain any changes you make in the comments though.
- Figure 4.52 (around page 468) in the CS:APP3e textbook illustrates the design of PIPE, which will be helpful.

9 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 three sets of files (for a total of six):
  - Part A: `kth_ll.ys`, `kth_bst.ys`, and `insert_ll.ys`.
  - Part B: `seq-full.hcl`.
  - Part C: `abscopy.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.

10 Installation & Usage Hints

- Experimental syntax highlighting files are provided for vim under `vim-y86-highlighting.tar.gz`. Extract this into your `~/.vim` folder and it should work directly. I recommend adapting this file to your own favorite editor to increase the amount of fun you have while doing the homework. Writing Y86-64 as plain text is plain suffering. This will also work on the ineks, of course.
- 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.
- 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 `Makefile`s 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 flex bison tcl-dev tk-dev
    ```
  - If you're on Mac (some of you probably are!), try out the experimental instructions in `macOS-compilation-instructions.pdf`.
  - Otherwise, or if these do not work, you should still be able to use the GUI remotely through the ineks. Read on.
- It is possible to connect to the ineks remotely and use the GUI of the simulator. First, connect to the `login` server (which allows X11 forwarding for now, unlike `external`):

  ```
  unix>  ssh -X -p 8085 eXXXXXXX@login.ceng.metu.edu.tr
  ```

  And then connect to an inek by using the -X parameter again:

  ```
  unix>  ssh -X inek42
  ```

  Afterwards you should be able to run the simulator in GUI mode over the connection. Since drawing commands are sent over the network with X11 forwarding, the GUI will take more time to get initialized than on your local machine.
- X11 Forwarding should work by default on Linux. For Mac and Windows, you will need to install an X Server. Examples that should work:
  - If you're using a Mac, install XQuartz, restart your computer (or logout/in) and you should be good to go (xQuartz should start running in the background on its own, or you can make it run). Make sure to check that your SSH configuration allows X11 forwarding if it does not work.
  - For Windows, assuming you already have PuTTY, you again need to install an X server like Xming. Once this is done, make sure that Xming is running in the background. Afterwards, enable X11 forwarding when connecting from PuTTY through Connection $->$ SSH $->$ X11. That should do it.
- 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.