Middle East Technical University | Department of Computer Engineering

CENG 331

Computer Organization

Fall '2024-2025

THE3: Architecture Lab

Optimizing the Performance of a Pipelined Processor

Due date: 19 December 2024, Thursday, 23:59

1 Introduction

In this lab, you will explore the design and implementation of a pipelined Y86-64 processor, focusing on optimizing both the processor and a benchmark program to achieve maximum performance. You can apply any semantics-preserving transformations to the benchmark program, make enhancements to the pipelined processor, or pursue both approaches. By the end of this lab, you will gain a deeper understanding of how the interplay between code and hardware influences program performance.

The lab is divided into three sections, each requiring a separate submission. In Part A, you'll write basic Y86-64 programs and gain familiarity with the Y86-64 tools. In Part B, you'll enhance the SEQ simulator by adding a new instruction. These two sections will build the foundation for Part C, the core of the lab, where you'll focus on optimizing both 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 two Y86-64 programs, whose required behavior is specified by the example C functions in `examples.c`. Ensure that you include your name and ID as a comment at the start of each program. You can test your programs by assembling them with the YAS assembler and running them with the YIS instruction set simulator.

When writing your Y86-64 functions, adhere to the x86-64 conventions for passing function arguments, utilizing registers, and managing the stack. This includes saving and restoring any callee-saved registers you modify.

`selection_sort_it.ys`: Iterative selection sort for the linked list

Write a Y86-64 program, `selection_sort_it.ys`, that iteratively sorts a given linked list in ascending order using the Selection Sort algorithm. Your program should include code to set up the stack structure, call a function, and then halt. The function, `selection_sort_it`, should be written in Y86-64 and must perform the same functionality as the C function `selection_sort_it` shown in Figure 1.

Your implementation should **rearrange the node links** to sort the linked list. Test your program using the provided 11-element linked list:

```
# Absolutely positioned here to debug addresses more easily.
.pos 0x200
head:
    .quad 17
    .quad node1
node1:
    .quad 24
    .quad node2
node2:
    .quad 6
    .quad node3
node3:
    .quad 11
    .quad node4
node4:
    .quad 4
    .quad node5
node5:
    .quad 5
    .quad node6
```

```
node6:
    .quad 3
    .quad node7
node7:
    .quad 40
    .quad node8
node8:
    .quad 19
    .quad node9
node9:
    .quad 52
    .quad node10
node10:
    .quad 8
    .quad 0 # Remember that 0 is null.
```

`selection_sort_rec.ys`: Recursive selection sort for the linked list

Write a Y86-64 program `selection_sort_rec.ys` that recursively sorts a given linked list in ascending order using the Selection Sort algorithm. Your program must implement recursion, following the structure of the C function `selection_sort_rec` shown in Figure 1.

Test your program using the same 11-element linked list as in `selection_sort_it`.

5 Part B

For this part, you will work in the directory `sim/seq`.

Your task in Part B is to extend the SEQ processor to support a new instruction, `cmpq`, a two-byte instruction of the form `cmpq rA, rB`. This instruction subtracts the value in register `rA` from the value in register `rB` and updates the condition codes accordingly. Note that it does not store the result.

To implement this instruction, you will modify the file `seq-full.hcl`, which contains the implementation of the SEQ processor as described in the CS:APP3e textbook. This file also includes declarations of constants that you will need for your solution.

Your HCL file must begin with a header comment containing the following information:

- Your name and ID.
- A description of the computations required for the `cmpq` instruction. Use the descriptions of `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, 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:

```
1 struct Node {
2     long data;
3     struct Node *head;
4 };
5
6 struct Node* selection_sort_it(struct Node* head)
7 {
8     if (head->next == NULL)
9          return head;
10
11     struct Node* sorted = NULL;
12     while (head != NULL) {
13         struct Node* max = head;
14         struct Node* prevMax = NULL;
15         struct Node* curr = head;
16         struct Node* prev = NULL;
17
18         while (curr != NULL) {
19             if (curr->data > max->data) {
21                 max = curr;
22                 prevMax = prev;
23             }
24             prev = curr;
25             curr = curr->next;
26         }
27
28         if (max == head)
29             head = head->next;
30         else
31             prevMax->next = max->next;
32
33         max->next = sorted;
34         sorted = max;
35     }
36      return sorted;
37 }
38
39 struct Node* selection_sort_rec(struct Node* head)
40 {
41     if (head->next == NULL)
42         return head;
43
44     struct Node* min = head;
45     struct Node* prevMin = NULL;
46     struct Node* curr;
47
48     for (curr = head; curr->next != NULL; curr = curr->next) {
49         if (curr->next->data < min->data) {
50             min = curr->next;
51             prevMin = curr;
52         }
53     }
54
55     if (min != head) {
56         struct Node* temp = head->next;
57         head->next = min->next;
58         if (temp == min)
59             min->next = head;
60         else {
61             min->next = temp;
62             prevMin->next = head;
63         }
64         head = min;
65     }
66
67     head->next = selection_sort_rec(head->next);
68
69     return head;
70 }
```


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

```
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 `cmpmany.yo` (testing `cmpq`) in TTY mode, comparing the results against the ISA simulation:

  ```
  unix>  ./ssim -t ../y86-code/cmpmany.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/cmpmany.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 process runs `ssim` on the benchmark programs to verify correctness by comparing the resulting processor state with the state produced by a high-level ISA simulation. Note that these programs do not test the newly added instructions; the goal is to ensure your modifications have not introduced errors into the existing instruction set. For more details, refer to the `../y86-code/README` file.

- *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 `cmpq`:

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

  To test your implementation of `cmpq`:

  ```
  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`).

6 Part C

For this part, you will work in the directory `sim/pipe`.

The `to_binary_string` shown in Figure 2 converts unsigned integers from an `n`-element integer array, `arr`, into binary string representations and stores them in a non-overlapping `buff` array. The function also returns the sum of the integers in `arr`. For example, converting the integer 3 should result in storing the binary string "00000011" in `buff`. You can assume that each element in `arr` will be in the range $[0, 255]$.

Figure 3 shows the baseline Y86-64 version of `to_binary_string`.

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

Your task in Part C is to optimize `to_binary_string.ys` and modify `pipe-full.hcl` to make `to_binary_string.ys` execute as efficiently as possible.

You will submit two files: `pipe-full.hcl` and `to_binary_string.ys`. Each file should begin with a header comment that includes the following information:

```
1 /*
2  * to_binary_string - Convert len values in arr to binary string and
3  * store in buff. Return sum of the values
4  */
5 word_t to_binary_string(word_t *arr, unsigned char **buff, word_t len)
6 {
7     word_t sum = 0;
8     word_t val;
9     int pow = 128;
10
11     while (len > 0) {
12         val = *arr++;
13         sum += val;
14         unsigned char * temp = buff;
15
16         for(pow = 128; pow > 0; pow >>=1){
17             if (val >= pow) {
18                 *temp++ = '1';
19                 val -= pow;
20             }
21             else
22                 *temp++ = '0';
23         }
24         len--;
25         buff++;
26     }
27
28     return sum;
29 }
```

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

```
 1 ##################################################################
 2 # to_binary_string.ys - Convert an arr block of len integers to
 3 # binary strings and store in buff.
 4 # Return the sum of integers contained in arr.
 5 # Include your name and ID here.
 6 # Describe how and why you modified the baseline code.
 7 ##################################################################
 8 # Do not modify this portion
 9 # Function prologue.
10 # %rdi = arr, %rsi = buff, %rdx = len
11 to_binary_string:
12 ##################################################################
13 # You can modify this portion
14         # Loop header
15         xorq %rax, %rax         # sum = 0;
16         irmovq $128, %rcx
17
18         andq %rdx,%rdx          # len <= 0?
19         jle Done                # if so, goto Done:
20 Loop:
21         mrmovq (%rdi), %r9      # read val from arr...
22         irmovq $8, %r8
23         addq %r8, %rdi          # arr++
24         addq %r9, %rax          # sum += val
25         rrmovq %rsi, %r10       # temp = buff
26         irmovq $128, %rcx       # pow = 128
27 inner_loop:
28         andq %rcx,%rcx          # pow <= 0?
29         jle updates
30         rrmovq %rcx, %r8
31         subq %r9, %r8
32         jg otw                  # check if val >= pow
33         subq %rcx, %r9          # val -= pow;
34         irmovq $49, %r8
35         rmmovq %r8, (%r10)      # *temp = '1'
36         jmp inner_updates
37 otw:
38         irmovq $48, %r8
39         rmmovq %r8, (%r10)      # *temp = '0'
40 inner_updates:
41         irmovq $1, %r8
42         addq %r8, %r10          # temp++
43         # all this for pow>>=1
44         xorq %r11, %r11
45 right_shift:
46         rrmovq %r8, %r12
47         subq %rcx, %r12
48         je shift_end
49         rrmovq %r8, %r11
50         addq %r8, %r8
51         jmp right_shift
52 shift_end:
53         rrmovq %r11, %rcx
54         jmp inner_loop
55 updates:
56         irmovq $1, %r8
57         subq %r8, %rdx           # len--
58         irmovq $8, %r8
59         addq %r8, %rsi           # buff++
60         andq %rdx,%rdx           # len > 0?
61         jg Loop                  # if so, goto Loop
62##################################################################
63 # Do not modify the following section of code
64 # Function epilogue.
65 Done:
66         ret
67 ##################################################################
68 # Keep the following label at the end of your function
69 End:
```

Figure 3: **Baseline Y86-64 version of the `to_binary_string` function.** See `sim/pipe/to_binary_string.ys`.

- Your name and ID.
- Provide a high-level explanation of your code modifications:
  - For `to_binary_string.ys`, describe the changes you made and the reasoning behind them. Use a step-by-step approach for clarity. For example: "I implemented X, which reduced the CPE from A to B.", "Next, I optimized Y, further reducing the CPE from B to C.".
  - For `pipe-full.hcl`, explain any modifications to the control logic, why you made them, and how they improved the performance of your code. If you choose not to modify `pipe-full.hcl` and focus solely on optimizing `to_binary_string.ys`, specify it.

Coding Rules

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

- Your `to_binary_string.ys` function must work for arbitrary array sizes. You might be tempted to hard-wire 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 `to_binary_string.ys` function must run correctly with YIS. By correctly, we mean that it must correctly convert the values of the first `n` elements of the `arr` list to binary strings *and* return (in `%rax`) the correct sum of the values. The linked list `arr` will contain more than `n` elements.
- The assembled version of your `to_binary_string` file must not be more than 1000 bytes long. You can check the length of any program with the `to_binary_string` function embedded using the provided script `check-len.pl`:

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

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

You are free to implement the `cmpq` instruction and modify the control logic if you believe it will improve performance, as long as your `pipe-full.hcl` implementation passes the regression tests (i.e., the base Y86-64 instruction set remains fully functional). However, there is no grade penalty if you choose not to modify `pipe-full.hcl`.

For the `to_binary_string.ys` function, you may apply any semantics-preserving transformations, such as reordering instructions, replacing groups of instructions with single instructions, removing unnecessary instructions, adding new instructions, or loop unrolling.

You are also allowed to add constant data (e.g., arrays, as done in Part A) to your program using the directives `.align`, `.quad` and `.pos`.

Building and Running Your Solution

To test your solution, you will need to build a driver program that calls your `to_binary_string` 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 `to_binary_string` 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 converting the values in the `arr` to binary strings.
- `ldriver.yo`: A *large driver program* that tests an `to_binary_string` 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 converting the values in the `arr` to binary strings.

Each time you modify your `to_binary_string.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 `to_binary_string.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 `to_binary_string.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 to_binary_string.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 to_binary_string is more than 1000 bytes long.

`0xdddd` : Some of the values in the `arr` was not converted and stored in the `buff`, correctly.

`0xeeee` : Some words were corrupted just before or after the destination region.

- *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 `cmpq` instruction, then to test `cmpq` 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 a total of 150 points: 40 points for Part A, 30 points for Part B, and 80 points for Part C.

Since this homework is more traditional than the Bomb and Attack Labs, your submissions will be checked for plagiarism as usual. Please remember that **we have a zero-tolerance policy for cheating**, which includes submitting work that is not your own, such as using sources from the internet.

Part A

Part A is worth 40 points, with 20 points awarded for each Y86-64 solution program. Each program will be evaluated for correctness, including proper handling of the stack and registers, and functional equivalence with the example C functions in `examples.c`.

Both programs will be considered correct if the graders do not identify any errors, the function correctly returns `0x260` in `%rax`, sorts the linked list properly, and does not corrupt other memory locations (except the stack) or exceed the bounds of the array. The required memory modifications are as follows:

```
Changes to memory:
0x0208: 0x0000000000000210 0x0000000000000280
0x0218: 0x0000000000000220 0x0000000000000270
0x0228: 0x0000000000000230 0x00000000000002a0
0x0238: 0x0000000000000240 0x0000000000000200
0x0258: 0x0000000000000260 0x0000000000000220
0x0268: 0x0000000000000270 0x0000000000000240
0x0278: 0x0000000000000280 0x0000000000000290
0x0288: 0x0000000000000290 0x0000000000000210
0x0298: 0x00000000000002a0 0x0000000000000000
0x02a8: 0x0000000000000000 0x0000000000000230
```

Part B

This part of the lab is worth 30 points:

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

Part C

This part of the Lab is worth 80 points: **You will not receive any credit if either your code for `to_binary_string.ys` or your modified simulator fails any of the tests described earlier.**

- 10 points each for your descriptions in the headers of `to_binary_string.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 `to_binary_string.ys` will be graded out of 20 if you do not. However, in case you do modify it explanations must be present.
- 60 points are allocated for performance. To receive credit, your solution must be correct, as defined earlier. Specifically, `to_binary_string` 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` (note that `cmpq` will not be tested during grading).

  Performance will be measured in terms 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 `to_binary_string` function running on the standard PIPE simulator with a large 64-element array requires 24727 cycles to copy 64 elements, resulting in a CPE of $24727/64 = 386.36$.

  Since some cycles are used to set up the call to `to_binary_string` and the loop within it, you may observe different CPE values for varying block lengths (typically, 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 `to_binary_string.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 `to_binary_string` function has CPE values ranging between 397.00 and 382.70, with an average of 384.76. Note that this Perl script does not check for the correctness of the answer. Use the script `correctness.pl` for this.

To earn 50 points for performance, you must achieve a threshold CPE of 146.5. If your average CPE is `c`, then your score `S` for this portion of the lab will be:

$$S = \begin{cases} 0 & c \leq 384 \\ 50 \cdot \frac{c-146.5}{384-146.5} & 146.5 \leq x \leq 384 \\ 50 & 146.5 \leq c \end{cases}$$

The remaining **10 points** will be scaled based on your CPE ranking between the highest CPE score of the top 3 students and the threshold CPE of 146.5. The top 3 students with the lowest CPEs (highest speedup) will receive a total of 60 points for Part C.

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

In Cases of Suspected Cheating

If any evidence of cheating is detected, you will be required to **attend a lab exam** where you will write a basic Y86-64 program. The exam date will be announced later and will have a duration of 1 hour.

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.
- 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 4.

  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.

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

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 `cmpq` will not be tested, you can change `pipe-full.hcl` to make `cmpq` 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 `cmpq`, 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: `selection_sort_it.ys`, and `selection_sort_rec.ys`.
  - Part B: `seq-full.hcl`.
  - Part C: `to_binary_string.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 hand-in files.

10 Installation & Usage 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.
- 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
    ```

- 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
  ```

  Afterward, 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. Afterward, 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.