Middle East Technical University | Department of Computer Engineering

CENG 331

Computer Organization

Fall '2020-2021

THE3: Architecture Lab

Optimizing the Performance of a Pipelined Processor

Due date: 27 December 2020, Sunday, 23:59

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. This includes saving and restoring any callee-saved registers that you use.

`rev.ys`: Iteratively reverse a linked list

Write a Y86-64 program `rev.ys` that iteratively reverses 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 (`rev`) that is functionally equivalent to the C `rev` function in Figure 1. Test your program using the following five-element list:

```
# A sample five-element linked list
# Aligned absolutely to make observing
# differences in the memory layout easier
.pos 0x100
ele0:
    .quad 0x0000a
    .quad ele1
ele1:
    .quad 0x000b0
    .quad ele2
ele2:
    .quad 0x00c00
    .quad ele3
ele3:
    .quad 0x0d000
    .quad ele4
ele4:
    .quad 0xe0000
    .quad 0
```

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

/* Reverse a linked list iteratively */
struct list *rev(struct list *head)
{
    struct list *prev = (void *) 0;
    while (head) {
        struct list *next = head->next;
        head->next = prev;
        prev = head;
        head = next;
    }
    return prev;
}

/* Reverse a linked list recursively */
struct list *rrev(struct list *head)
{
    struct list *rev_head;
    if (!head || !head->next)
        return head;
    rev_head = rrev(head->next);
    head->next->next = head;
    head->next = (void *) 0;
    return rev_head;
}

/* Copy overlapping data with a checksum */
long move(long *dst, const long *src, long len)
{
    long dst_v = (long) dst;
    long src_v = (long) src;
    long checksum = 0;
    long step = 1;
    long elem_size = sizeof(long);
    if (src_v < dst_v && dst_v < src_v + elem_size * len) {
        dst = dst + len - 1;
        src = src + len - 1;
        step = -1;
    }
    while (len > 0) {
        checksum ^= *src;
        *dst = *src;
        dst += step;
        src += step;
        len--;
    }
    return checksum;
}
```


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

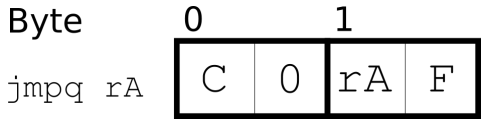

Figure 2: **Encoding of the jmpq instruction**

`rrev.ys`: Recursively reverse a linked list

Write a Y86-64 program `rrev.ys` that recursively reverses a linked list. This code should be similar to the code in `rev.ys`, except that it should use a function `rrev` that recursively reverses the given list, as shown with the C function `rrev` in Figure 1. Test your program using the same five-element list you used for testing `rev.ys`.

`move.ys`: Copy a source block to a destination block

Write a program (`move.ys`) that copies a block of words from one part of memory to another (possibly overlapping) area of memory, computing the checksum (xor) of all the words copied.

Your program should consist of code that sets up a stack frame, invokes a function `move`, and then halts. The function should be functionally equivalent to the C function `move` shown in Figure Figure 1. Test your program using the following ten-element array, with `arrayp3` as the destination pointer, `array` as the source pointer and a size of 5:

```
# an array of 9 elements
# again, positioned absolutely
.pos 0x100
array:
    .quad 0x000000001
    .quad 0x000000020
    .quad 0x000000300
# a pointer to the fourth element here
arrayp3:
    .quad 0x000004000
    .quad 0x000050000
    .quad 0x000600000
    .quad 0x007000000
    .quad 0x080000000
    .quad 0x900000000
```

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, `jmpq`. Observe that Y86-64 only supports conditional and unconditional jumps to *constant* addresses. You have already seen in the Bomb Lab that being able to jump to a runtime address stored in a register is useful for implementing jump tables in x86-64. To extend Y86-64 with this capability, you will add the new two-byte instruction `jmpq rA`, which will jump to the address stored in register `rA` when executed. The encoding of `jmpq` is shown in Figure 5. The main takeaway is that only the first half-byte of the register byte is used while the second one is ignored, just like `pushq` and `popq`.

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:

- Your name and ID.
- A description of the computations required for the `jmpq` 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 `jtable.yo` or `jsimple.yo` (testing `jmpq`) in TTY mode, comparing the results against the ISA simulation:

  ```
  unix>  ./ssim -t ../y86-code/jtable.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/jtable.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 `jmpq` and `leave`:

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

  To test your implementation of `jmpq`:

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

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

```
1 /*
2  * ncopy - copy src to dst, returning number of positive ints
3  * contained in src array.
4  */
5 word_t ncopy(word_t *src, word_t *dst, word_t len)
6 {
7     word_t count = 0;
8     word_t val;
9
10    while (len > 0) {
11        val = *src++;
12        *dst++ = val;
13        if (val > 0)
14            count++;
15        len--;
16    }
17    return count;
18 }
```

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

6 Part C

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

The `ncopy` function in Figure 3 copies a `len`-element integer array `src` to a non-overlapping `dst`, returning a count of the number of positive integers contained in `src`. Figure 4 shows the baseline Y86-64 version of `ncopy`. 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 `ncopy.ys` and `pipe-full.hcl` with the goal of making `ncopy.ys` run as fast as possible.

You will be handing in two files: `pipe-full.hcl` and `ncopy.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 `ncopy.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. Note that you can also choose to not modify `pipe-full.hcl` at all and keep your optimizations restricted to the code.

To help with optimization, the PIPE processor description provided in `pipe-full.hcl` comes extended with an extra instruction `iaddq C, rB`, that adds a constant value `C` to register `rB`, described in Homework problems 4.51 and 4.52.

Coding Rules

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

```
1  ##################################################################
2  # ncopy.ys - Copy a src block of len words to dst.
3  # Return the number of positive words (>0) contained in src.
4  #
5  # Include your name and ID here.
6  #
7  # Describe how and why you modified the baseline code.
8  #
9  ##################################################################
10 # Do not modify this portion
11 # Function prologue.
12 # %rdi = src, %rsi = dst, %rdx = len
13 ncopy:
14
15 ##################################################################
16 # You can modify this portion
17         # Loop header
18         xorq %rax,%rax          # count = 0;
19         andq %rdx,%rdx          # len <= 0?
20         jle Done                # if so, goto Done:
21
22 Loop:   mrmovq (%rdi), %r10     # read val from src...
23         rmmovq %r10, (%rsi)     # ...and store it to dst
24         andq %r10, %r10         # val <= 0?
25         jle Npos                # if so, goto Npos:
26         irmovq $1, %r10
27         addq %r10, %rax         # count++
28 Npos:   irmovq $1, %r10
29         subq %r10, %rdx         # len--
30         irmovq $8, %r10
31         addq %r10, %rdi         # src++
32         addq %r10, %rsi         # dst++
33         andq %rdx,%rdx          # len > 0?
34         jg Loop                 # if so, goto Loop:
35 ##################################################################
36 # Do not modify the following section of code
37 # Function epilogue.
38 Done:
39         ret
40 ##################################################################
41 # Keep the following label at the end of your function
42 End:
```

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

- Your `ncopy.ys` 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.ys` 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.
- The assembled version of your `ncopy` file must not be more than 1000 bytes long. You can check the length of any program with the `ncopy` function embedded using the provided script `check-len.pl`:

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

- Your `pipe-full.hcl` implementation must pass the regression tests in `../y86-code` and `../ptest` (without the `-i` and `-j` flag that test `iaddq` and `jmpq`, respectively).

Other than that, you are free to implement the `jmpq` 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 `ncopy.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 `ncopy` 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 `ncopy` function on small arrays with 4 elements. If your solution is correct, then this program will halt with a value of 2 in register `%rax` after copying the `src` array.
- `ldriver.yo`: A *large driver program* that tests an `ncopy` function on larger arrays with 63 elements. If your solution is correct, then this program will halt with a value of 31 (`0x1f`) in register `%rax` after copying the `src` array.

Each time you modify your `ncopy.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 `ncopy.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 `ncopy.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 ncopy.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 count

  `0xcccc` : Function ncopy 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 `jmpq` instruction, then to test `iaddq` and `jmpq` along with all the standard instructions:

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

- *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: 30 points for Part A, 35 points for Part B, and 100 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. Remember that **we have a zero tolerance policy for cheating**.

Part A

Part A is worth 30 points, 10 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 `rev.ys` and `rrev.ys` will be considered correct if the graders do not spot any errors in them, and their respective `rev` and `rrev` properly reverse the provided linked list in memory, returning the new head `0x140` in register `%rax`.

The program `move.ys` will be considered correct if the graders do not spot any errors in them, and the `move` function returns the checksum `0x54321` in register `%rax`, moves the first five elements of the provided array three indices forward, and does not corrupt other memory locations.

Part B

This part of the lab is worth 35 points:

- 10 points for your description of the computations required for the `jmpq` instruction.
- 10 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 `jmpq`.

Part C

This part of the Lab is worth 100 points: **You will not receive any credit if either your code for `ncopy.ys` or your modified simulator fails any of the tests described earlier.**

- 20 points each for your descriptions in the headers of `ncopy.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 `ncopy.ys` will be graded out of 40 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, `ncopy` runs correctly with YIS, and `pipe-full.hcl` passes all tests in `y86-code` and `ptest` (remember that `iaddq` and `jmpq` 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 `ncopy` function running on the standard PIPE simulator with a large 63-element array requires 897 cycles to copy 63 elements, for a CPE of $897/63 = 14.24$.

  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. You can use the Perl script `benchmark.pl` in the `pipe` directory to run simulations of your `ncopy.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 `ncopy` function has CPE values ranging between 29.00 and 14.27, with an average of 15.18. Note that this Perl script does not check for the correctness of the answer. Use the script `correctness.pl` for this.

  You should be able to achieve an average CPE of less than 9.00. If your average CPE is c, then your score S for this portion of the lab will be:

$$S = \begin{cases} 0\,, & c > 10.5 \\ 20 \cdot (10.5 - c)\,, & 7.50 \leq c \leq 10.50 \\ 60\,, & c < 7.50 \end{cases}$$

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

8 Bonus Opportunities

Since the homework is not exactly trivial, there are three opportunities to get extra points from the homework for those who get really into it, none of them being mutually exclusive. Each will grant you one extra point, up to a total of three. Since the homework constitutes six points of the course grade, you can go up to nine with the bonuses: possibly 150% of the maximum grade just like the Attack Lab.

Performance

If your code achieves an average CPE below the maximum grade threshold 7.50, i.e. `benchmark.pl` shows an average CPE ≤ 7.49, and you have properly explained how you've achieved this performance, you get one extra point.

Performance++

If your code further achieves an average CPE below 7.40, i.e. `benchmark.pl` shows an average CPE ≤ 7.39, and the modifications that led to this are explained clearly, you get one more point!

`jmpq` on PIPE

Even though implementing `jmpq` on SEQ is relatively simple, implementing the same instruction properly on PIPE is more difficult. This is due to two main issues:

- `jmpq` has a data dependency, it needs to jump to the last value in the target register, which may not yet have been written-back to the register file.
- `jmpq` causes a control issue, since the next instruction cannot be fetched before the address has been extracted from the target register.

If you manage to overcome these problems and implement `jmpq` on PIPE, explaining how you did it in the comments, you will get an extra point.

Note that a possible solution would be stalling `jmpq` at the fetch stage for three or four cycles. This is not a proper solution and would seriously disrupt the pipeline. A proper (and thus, acceptable) solution will:

- Pass all the regression tests, including the `jmpq` instruction tests:

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

- Run `jsimple.ys` in 5 cycles (as reported by your PIPE's simulator `psim`):

  ```
  unix> ./psim -t ../y86-code/jsimple.yo
  ```

9 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 unnecessarily 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 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.

```
    .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 256 to be far enough from
# your initial code to not have problems
.pos 0x100
    # .. 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 768,
# far away from the code, your code
# should not be long enough to get here anyway!
.pos 0x300
stack:
```

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

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!
- Be careful with `iaddq`! Even though adding values to registers directly is very convenient, each `iaddq` is 10 bytes (due to the value being stored in the instruction) and might cause you to go through your 1000 byte program size limit rather quickly.
- 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 `iaddq` will not be tested (as well as `jmpq` unless you're going for the bonus), you can change `pipe-full.hcl` to make `iaddq` or `jmpq` 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 `jmpq` or `iaddq`, and you should perform the regression tests without the `-i` and `-j` flags. But this is fine since they will not be tested during grading! You definitely do not need to do this kind of thing to achieve maximum performance (below 7.40 CPE), this explanation is only here in case you want to do such a thing and are wondering: "Am I allowed to do it?". 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 might 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 three sets of files (for a total of six):
  - Part A: `rev.ys`, `rrev.ys`, and `move.ys`.
  - Part B: `seq-full.hcl`.
  - Part C: `ncopy.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 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 editor to make writing Y86-64 more fun than writing plain text.

- 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 (I know some of you 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 may take more time to get initialized than on your local machine.
- 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.