Computer Organization and Networks Practicals 2024/25

October 15, 2024

Contents

-1 Introduction

This document describes the tasks of the course "Computer Organization and Networks Practicals" for the winter term 2024/25. In this course, we are going to study computer architectures and networking stacks. We will discuss how hardware is designed using the hardware description language SystemVerilog. In the network part, we look at QUIC, the protocol powering the next generation of the internet.

-1.1 Assignment Sheet

The main purpose of the assignment sheet is to specify what you need to do to receive a positive grade at the end of the semester. The assignment sheet (you are reading right now) can be found online (a link is also provided on the course website) in the upstream git repository at

`https://extgit.iaik.tugraz.at/con/upstream-2024`

This repository provides the latest version of the assignment. When providing an update of the assignment, we will push a new version to the upstream repository and will also announce it in `#con`.

-1.2 Communication Channels

We provide the following communication channels:

CON Email. We provide the email address con@iaik.tugraz.at for personal requests. Use this email only if you have a question which cannot be discussed publicly.

Discord. Discord is used as a primary channel to reach your tutors and ask questions besides question hours. You need to register an account on Discord and then join the "IAIK" server. If you pick a username related to your legal name, it helps your Teaching Assistant (TA) to recognize you. To join at Discord, you can use the following invitation link:

`https://discord.gg/mxuUnjP`

In the channel `#getting-started`, react with 📶 to the message so the bot adds you to the corresponding `#con` channels.

- The CON team uses `#con-announcements` to make short announcements regarding exercises and assignment interviews. Make sure to stay up to date by checking this channel from time to time.
- `#con-questions` is a forum channel for all CON participants to ask questions in textual form. Be kind to other participants and be aware that you are not allowed to post solutions to exercises. It serves as a place of discourse between students and TAs, which might contribute but do not have to answer questions here.
- The text channel `#con` is a relic of past CON exercises, which we decided to keep as an archive for you to look at. Feel free to check it out, but stick to the `#con-announcements` channel for new questions.
- `#con-ta` is a prefix used for audio-only channels, one per TA. If you ever need a voice channel for you and your TA (e.g. in case of a remote assignment interview), this is the right place.

-1.3 Tutorial Videos

Date	Content	Video
2023-10-06	Getting started	TU Graz Cloud
2023-10-06	Task 1	TU Graz Cloud
2023-10-06	Task 2	TU Graz Cloud
2023-10-06	Task 3	TU Graz Cloud
2023-10-06	Task 4	TU Graz Cloud
2023-10-06	Task 5	TU Graz Cloud

Table 1: Tutorial session videos.

Tutorial videos are prerecorded videos (c.f. Table 1) to be published at the start of the semester and will be shared on the TU Graz cloud. The main goal is to introduce you to the corresponding task and show you the required toolchain.

-1.4 Toolchain

The toolchain is introduced in the tutorial videos (Section -1.3), but we still want to document it here once more. Here is a list of the entire software stack relevant for the practicals:

- git for version control (and a GitLab server for submissions)
- SystemVerilog (→ IEEE 1800-2017)
- SV2V to convert SystemVerilog to Verilog (→ sv2v)
- Yosys for synthesis (→ Yosys Open SYnthesis Suite)
- Icarus Verilog (iverilog) v12.0 for SystemVerilog simulation (→ GitHub project)
- GTKWave for debugging (→ GTK+ based wave viewer)
- RISC-V (→ RISC-V ISA specification)

- asmlib, a python library (also on PyPI), to simulate RISC-V execution cross-platform in python

In our build scripts, we provide `Makefiles` which can be run with GNU Make. bash scripts are also going to be used. As we don't want to bother you with installing software, we provide a Virtual Machine (VM) for VirtualBox. The VM has the entire software stack preinstalled and is based on Ubuntu 24.04:

`https://cloud.tugraz.at/index.php/s/oLtADP6cLsTtMCj`

The user for this VM is `convm` and the password `convm`.

-1.5 Question Hours

Question hours take place in person during specific timeslots. During question hours, two or more TAs will be present to answer your questions regarding the assignments. This should be your primary place to ask questions, as we do not require our TAs to answer questions on Discord. There will be three question hour slots in every week with a CON-related deadline.

For the time and location of question hours, please refer to your calendar in TUG-Online.

-1.6 Submissions

At the beginning of the semester, you will receive account credentials for a GitLab instance. Using git, you can submit your deliverables in your personal git repository. When submitting, pay attention to the following aspects:

- You need to *tag* your commit. The tag name format is `submission-task-x`, where `x` corresponds to the respective task number, excluding the period. For example, the git tag for the submission of Task 1 is `submission-task-1`.
- After tagging the commit, don't forget to push your tag with '`git push --tags`'!
- You can check the state of your git repository by visiting your git repository in GitLab.
- If you tagged the wrong commit, you can delete the tag and tag the correct commit.

The hard deadlines for the respective tasks are given in Table 2. If you submit your solution after a deadline, you will get a deduction of 8 points for each late day (every 24h) on the respective task.

-1.7 Task Interviews

There will be two task interviews. The first interview will cover tasks 1, 2 and 3. The second will cover tasks 4 and 5. The dates for the interviews will be organized by your TA and they will inform you ahead of time. The interviews include general questions on each topic and also a discussion of your submitted solution. The goal of the interviews is

Task	**Deadline**	**Max. points**
Task 0 Getting Started	We, 2024-10-16 23:59	max. 1 point
Task 1 Simple Calculator	Th, 2024-10-31 23:59	max. 14 points
Task 2 Cordic Accelerator	Fr, 2024-11-15 23:59	max. 20 points
Task 3 CPU Integration	Fr, 2024-11-29 23:59	max. 15 points
Task 4 XGCD in RISC-V	Fr, 2024-12-20 23:59	max. 20 points
Task 5 QUIB	Fr, 2025-01-17 23:59	max. 30 points

Table 2: Task submission deadlines and maximum achievable points.

to verify you did the implementation yourself, understood the topic, and collect feedback on both sides.

-1.8 Grading

The maximum points per task are listed in Table 2. Depending on your achieved points, you will get a grade according to the following scheme.

0–50	Points	→	Nicht genügend (5)
51–62	Points	→	Genügend (4)
63–75	Points	→	Befriedigend (3)
76–87	Points	→	Gut (2)
88–100	Points	→	Sehr gut (1)

-1.9 Plagiarism

We will regularly check all submissions using automated plagiarism checking tools. If we detect a case of plagiarism, all involved people (the source and all sinks) will receive the grade `U` (Ungültig/Täuschung). Please also refer to the lecture slides for further information. Cases of plagiarism are handled as soon they are detected.

To avoid getting into a situation of plagiarism follow the following rules:

- Don't share code!
- Don't tell/dictate your solution to others!
- Don't copy code from the internet!
- Commit regularly to show how you solved the task!

-1.10 AI Usage

We treat the involvement of ChatGPT and similar tools the same way as the involvement of another natural person. That is, for involvement that qualifies as plagiarism or an impermissible level of assistance, the consequences will be the same in both cases to the strictest extent possible.

0 Task 0: Getting Started

The goal of this task is to understand good scientific practice and to get familiar with the CON toolchain (*i.e.*, the virtual machine and SystemVerilog).

0.1 Understanding Scientific Practice

Before you get started, check the documents we provide on plagiarism and watch the corresponding video for this task. It is part of Task 0 to read the plagiarism documents and to acknowledge them.

1. Understand what plagiarism is. You can find more information on this topic on plagiarism.org.
2. Study the document "Guidelines on Safeguarding Good Scientific Practice".
3. Understand the consequences described in Section -1.9 and in the lecture slides.

0.2 Specification

The goal of this task is to set up the virtual machine with the CON toolchain and start a small hardware simulation written in SystemVerilog. Therefore, download the CON virtual machine image and install the program VirtualBox. After successfully adding the VM image, you should clone your CON working repository. In the `task-0` folder, you should find a file called `lfsr.sv`. This file contains the implementation of a 32-bit linear feedback shift register (LFSR) with an asynchronous reset. Figure 1 visualizes this LFSR. Read the SystemVerilog code and try to understand how it works. When reset, the register should hold your matriculation number. Encode it like this: `"32'd 0123456"`. Assign the value of your LFSR to the output `lfsr_state_o`. After you implemented the LFSR, run the `'make'` command to generate a pseudo-random magic number based on your matriculation number. Run `'make reference MATR_NUM=<your matriculation number>'` to generate a reference value. Your magic number and the reference should be identical.

Moreover, you should create a document that states that you understand plagiarism and the consequences of handing in plagiarism.

0.3 Deliverables

All files must be submitted in folder `task-0` of your repository. All files of the upstream repository must be included!

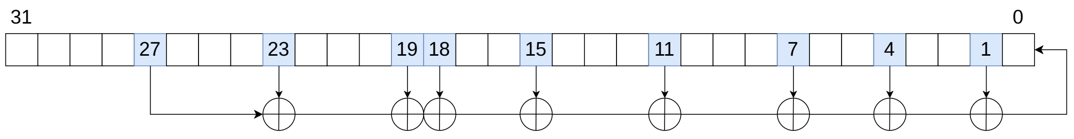

Figure 1: Design of the LFSR.

1. After reading content according to Section 0.1, create a text file with the name `scientific_practice.txt` and write a statement that
   - you understood what plagiarism is,
   - you acknowledge the consequences,
   - and that you won't submit plagiarized work.

   Add this file to your git repository.

2. Create a `README.md` file and describe which parts of your submission are (in-)complete to give your TA an overview.

3. Submit your modified SystemVerilog module for the LFSR (with your matriculation number) in `lfsr.sv`.

Add all files to the git repository. Make sure to commit your files and push them to your git repository on GitLab. Also, don't forget to create a tag and push it according to Section -1.6.

1 Task 1: Simple Calculator

This task is designed to give you an introduction to hardware design in SystemVerilog. You will implement combinational logic that is capable of adding and subtracting 5-bit values and then combine it with a state machine to build a small calculator.

1.1 Design

The hardware design for the calculator is organized in a hierarchy. You will implement the individual modules in separate files and instantiate them in other modules. The file structure and the interfaces of the modules are given.

1.1.1 Full-Adder

We will first start with a basic building block of digital logic: the full-adder. It has two regular inputs `a_i` and `b_i` and an additional input `c_i` for a carry bit. The output is comprised of the sum of the two inputs `sum_o` and a carry output `c_o`. The truth table for the circuit is given in Table 1.1. Implement the combinational logic that realizes this truth table in the file `fulladder.sv`.
Note: Do not use SystemVerilog's arithmetic operators (e.g. `'+'`). Implement the full-adder using only Boolean logic gates.

a_i	b_i	c_i	s_o	c_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 1.1: Truth table of a full-adder

1.1.2 Add-Sub-Machine

Next, we will integrate several instances of the full-adder circuit from the previous section to implement an add-sub-machine that is capable of adding and subtracting 5-bit values. The interface for this module is given in the file `addsub.sv`. To achieve addition with more than one bit, you will need to chain together several instances of your full-adder. In addition, the `mode_i` input controls whether the machine performs addition or subtraction. Use your knowledge about Two's complement to extend the addition circuit with subtraction capabilities.

1.1.3 Calculator

Now that we have a functional add-sub-machine we can finally build a small calculator with it. For this, we create a finite state machine that controls the behavior and integrate the add-sub-machine as a datapath. The state diagram for the state machine is given in Figure 1.1. While the `calc_i` input stays low, the calculator alternates between the *LOAD_A* and *LOAD_B* states whenever it reads a valid value, indicated by `input_valid_i`. Operands, which are read from the module's input, must be saved in registers depending on the current state and the validity. For example, whenever the state machine is in state *LOAD_A*, the `input_valid_i` signal is high and a rising clock edge occurs, the input `a_i` should be stored in an internal register. The `calc_i` signal will move the state machine into the *CALC* state, where it remains until the `clear_i` signal resets the calculator. Note that the `clear_i` signal will always reset the state machine to the *LOAD_A* state and reset the operand registers. The `output_valid_o` signal is only set when the state machine is in the *CALC* state. While the calculators operands should be stored in registers, the output of the add-sub-machine is directly connected to the `result_o` signal. This means, the output of the calculator is directly influenced by the `mode_i` signal, independent of the state.

The given testbench in `tb/calc_tb.sv` only performs simple tests that load 2 numbers, calculate a result and reset the state machine. Read the testbench to understand how it works. To test the full functionality of your implementation, consider extending it with additional functions (SystemVerilog tasks).

1.2 Deliverables

All files must be submitted in folder `task-1` of your repository.

1. Edit the `README.md` file and describe which parts of your submission are complete to give your TA an overview.

2. Implement the combinational logic of a full-adder in `src/fulladder.sv`.

3. Use the full-adder circuit to build an add-sub-machine in `src/addsub.sv`.

4. Integrate the add-sub-machine as a datapath into the calculator and control it with the presented finite state machine. Add your implementation in `src/calculator.sv`.

Add all these files to the git repository. Make sure to commit your files and push them to your git repository on GitLab. Also, do not forget to create a tag and push it according to Section -1.6.

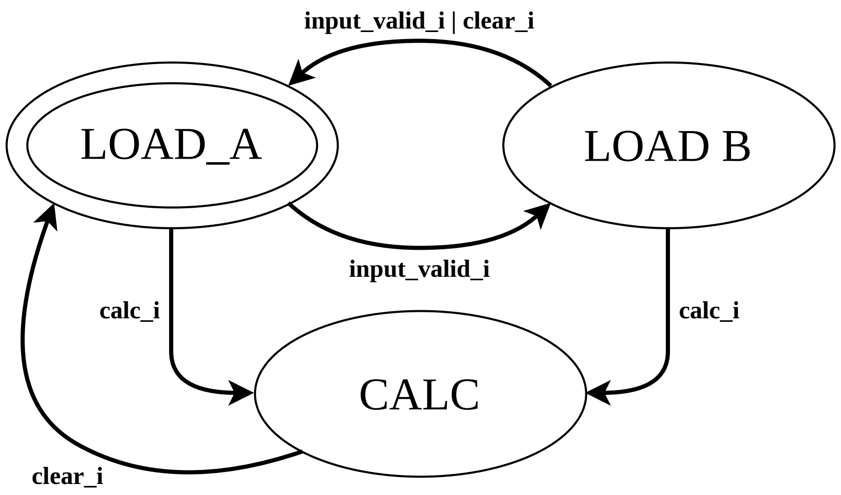

Figure 1.1: The finite state machine of the calculator.

1.3 Hints

1. As the provided testbench is not verifying the entire specified behavior, extend the testbench found in `tb/calc_tb.sv` to cover everything.
2. Run `make lint` to statically analyze your SystemVerilog design for errors.
3. Run `make test` to test your implementation with all testvectors found in the `test` directory.
4. Run `make run TARGET=<testvector>` to run hardware simulation with a single testvector.
5. Run `make view TARGET=<testvector>` to view simulation traces for a certain testvector in GTKwave.
6. Run `make synth` to synthesize your design with Yosys and check for parasitic latches.

2 Task 2: Cordic Accelerator

In this task you will implement a register-transfer-level (RTL) model of a hardware accelerator for the Cordic algorithm. The abbreviation Cordic stands for "COordinate Rotation DIgital Computer", published by Jack E. Volder in 1959. The Cordic algorithm can be used to bypass the computation of the square root of the inverted tangent, but also many other functions. The beauty of this algorithm is that it only requires additions and shift operations to compute trigonometric functions. The accelerator you will build uses an iterative algorithm to approximate sine and cosine values of a given angle. Start by reading the explanation of the Cordic algorithm in the following section.

2.1 Cordic Algorithm Description

In our case, the Cordic algorithm is used in the so-called rotation mode, where the following vector is rotated.

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$$

We can rotate the vector (x_0, y_0) counter-clockwise with the angle ϕ by multiplying it with a rotation matrix. We receive the result in the rotated vector (x_1, y_1). If you factor out $\cos(\phi)$ from the rotation matrix, only one trigonometric function remains in the matrix.

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \cos(\phi) \cdot \begin{bmatrix} 1 & -\tan(\phi) \\ \tan(\phi) & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \tag{2.1}$$

However, instead of performing a single rotation to an **arbitrary** target angle ϕ, the same result can also be approximated by performing a sequence of smaller rotations using **fixed** angles α_i that eventually converge to the target angle. This idea is depicted in Figure 2.1 where the initial vector V_0 gets rotated to V_1 (e.g., 45°), which then gets rotated to V_2 (e.g., 67.5°) and subsequently gets rotated to V_3 (e.g., 56.25°). The angle ϕ is basically decomposed into a sum of known angles α_i.

$$\phi = \sum_i \alpha_i$$

As the result, computation of the rotation in Equation (2.1) can be approximated with Cordic by iteratively applying Section 2.1 and Section 2.1. In each iteration the vector is rotated either clockwise or counter-clockwise. Depending on the rotation direction, only the sign needs to be changed: An addition changes to a subtraction and vice versa. By

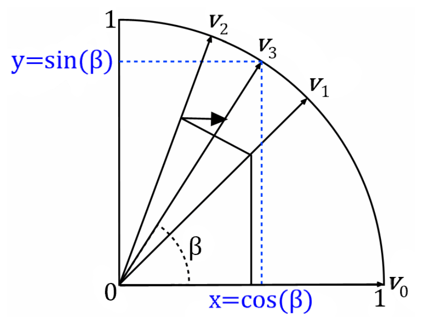

Figure 2.1: Cordic: Vector iteratively converges to target angle β. Source

choosing the proper rotation direction in every iteration, the vector eventually converges to the original target angle ϕ.

$$x_{i+1} = \cos(\alpha_i) \cdot (x_i - \tan(\alpha_i) \cdot y_i) \tag{2.2}$$
$$y_{i+1} = \cos(\alpha_i) \cdot (y_i + \tan(\alpha_i) \cdot x_i) \tag{2.3}$$

Optimizing the multiplication with $\tan(\alpha_i)$. By selecting the angles α_i in a smart way it is possible to further optimise the implementation. In particular, for the Cordic algorithm the angles α_i are selected such that $\tan(\alpha_i) = 2^{-i}$ (*i.e.*, 1, 1/2, 1/4, 1/8, ...). Subsequently, multiplication with $\tan(\alpha_i)$ can be implemented using a simple arithmetic right-shift operation by i positions. The resulting iteratively applied equations look as follows:

$$x_{i+1} = \cos(\alpha_i) \cdot (x_i - (y_i >> i))$$
$$y_{i+1} = \cos(\alpha_i) \cdot (y_i + (x_i >> i))$$

Listing 1 `angles.c`: Compute and print the first 29 angles α_i.

```c
#include <stdio.h>
#include <math.h>

void print_angles(void) {
  for (int i = 0; i < 29; ++i) {
    double angle = atan(1.0 / (1 << i));
    printf("%d: \t %.10f\n", i, angle);
  }
}

int main(void) {
  print_angles();
  return 0;
}
```

The C-program in Listing 1 computes and prints the first 29 values for α_i in floating point notation. To compile the program, invoke the compiler on the command line using `gcc -o angles angles.c -lm` and run it by executing `./angles`. After conversion into fixed-point representation (see Section 2.1.2), these constants can be directly stored within a lookup table in the C and assembler programs.

Optimizing the multiplication with $\cos(\alpha_i)$. Naively implementing Cordic still requires two multiplications with $\cos(\alpha_i)$ in every iteration. However, even a single multiplication is still expensive, especially if the instruction set of the processor does not have an instruction for this operation. Fortunately, these multiplications can be combined into a single constant K which is computed as the product of all the individual $\cos(\alpha_i)$.

$$K = \prod_{i=0}^{n-1} \cos(\alpha_i)$$

This constant K can then be multiplied with the final x_i and y_i when the unit vector $V_0 = (1, 0)$ was used initially. Alternatively, starting with a scaled unit vector $V_0 = K \cdot (1, 0) = (K, 0)$ removes the need for performing multiplications as part of Cordic completely. Since our Q3.28 fixed-point representation has a constant resolution, we can calculate K with the $\alpha_i's$ that we precomputed using Code 1. The value of K we use is 0.607252935. (Hint: also needs to be converted to Q3.28 according to Section 2.1.2)

The pseudo code for the final Cordic algorithm in rotation mode for angles between -90 and 90 degrees, is shown in Algorithm 1. Here, we rotate the K-scaled unit vector $(x_0 = K, y_0 = 0)$ to the defined angle ϕ, for which we want to compute the sine and cosine. In every iteration, we converge to the angle ϕ until the difference is almost zero. Then the computed x-component equals to the *cosine* and the y-component the *sine* of the angle ϕ.

2.1.1 Supporting all quadrants

Algorithm 1 only supports angles between -90 and 90 degrees or $-\pi/2$ to $\pi/2$. In order to extend the range to $\pm 2\pi$, we slightly need to modify the Cordic algorithm. Hereby, you need the exploit the symmetry properties of sine and cosine. Hint: Draw a unit circle to find the necessary pre- and post-processing steps. You must extend the algorithm such that it works for the domain $\pm 2\pi$.

2.1.2 Q3.28 Fixed-point Number Format

The RISC-V RV32I compatible processor we are using does not support floating point numbers natively, *i.e.*, all registers contain integer values. For the computation of the Cordic algorithm, we need a decimal number. For this, we use the so-called fixed-point number representation, the Qm.n number format. In contrast to a floating point number, the fixed-point number format has a constant resolution over the whole domain.

Qm.n uses m-bits for the integer part, n-bits for the decimal part, and 1-bit for the sign. In this assignment, we use the *Q3.28* number format. With this we get a domain

Algorithm 1 Cordic algorithm pseudo code

```
$x_{acc} \leftarrow K$
$y_{acc} \leftarrow 0$
$\theta_{acc} \leftarrow$ angle
$i \leftarrow 0$
**for** $i < 29$ **do**
    $x_{shift} \leftarrow y_{acc} >> i$
    $y_{shift} \leftarrow x_{acc} >> i$
    **if** $\theta_{acc} <= 0$ **then**
        $x_{acc} \leftarrow x_{acc} + x_{shift}$
        $y_{acc} \leftarrow y_{acc} - y_{shift}$
        $\theta_{acc} \leftarrow \theta_{acc} + \alpha_i$
    **else**
        $x_{acc} \leftarrow x_{acc} - x_{shift}$
        $y_{acc} \leftarrow y_{acc} + y_{shift}$
        $\theta_{acc} \leftarrow \theta_{acc} - \alpha_i$
    **end if**
    $i \leftarrow i + 1$
**end for**
sin $\leftarrow y_{acc}$
cos $\leftarrow x_{acc}$
```

ranging from -8 to 7.99999999627471 and a constant resolution of $\frac{1}{2^{28}}$. This is enough to cover angles between -2π to 2π. The conversion between a decimal number $nr_{decimal}$ and a fixed-point number $nr_{Q3.28}$ works as follows:

$$nr_{Q3.28} = \text{round}\left(nr_{decimal} \cdot 2^{28}\right)$$

If you want to convert the number 1.23 to Q3.28, we compute:

$$\text{round}\left(1.23 \cdot 2^{28}\right) = 330175611 = \texttt{0x13ae147b}$$

The multiplication can also be carried out using a left-shift operation. For negative numbers, the twos-complement representation comes into play. Therefore, you need to store the result in a *signed* datatype. For example, the negative number -1.23 gets converted the following way:

$$\text{round}\left(-1.23 \cdot 2^{28}\right) = -330175611 = \texttt{0xec51eb85}$$

For the opposite computation, if you want to convert a fixed-point number back into a decimal number, you have to divide by 2^{28}. For this, you can also use the *arithmetic* shift operation by 28 positions to the right.

2.2 Design

We provide you with a skeleton for your Cordic implementation, which you can find in `src/cordic.sv`. It includes the interface (see Listing 2.1), as well as precomputed parameters such as K, α_i and multiples of π in Q3.28 format. Implement a finite state machine to control the execution of the algorithm. Your implementation may take any (reasonable) number of cycles to finish. The signal `start_i` should start the calculation. While the state machine is performing calculations, set the `busy_o` signal to 1. When the calculation is finished and the result is visible on the `result_o` output, signal this by setting the `valid_o` output to 1. The result and the `valid_o` signal must stay valid until a new calculation is started. We also provide a testbench in `testbench/testbench_cordic.sv` and a testvector in `test/student.testvec`. You can add more testvectors by adding your own files (inputs & expected outputs) in this directory. **Important:** The provided testvector is **not** exhaustive, meaning you will need to add your own testcases to verify the full expected functionality of your Cordic accelerator. Each line in a testvector consists of the input angle in hexadecimal representation and the control signal `sine_cosine_i`, 0 for sine and 1 for cosine. Refer to the example test vectors from Section 2.5 for a detailed description.

Listing 2.1: Cordic top-level interface in Verilog.

```
module cordic(
  input  logic        clk_i,
  input  logic        reset_i,
  input  logic [31:0] angle_i,
  input  logic        sine_cosine_i,
  input  logic        start_i,
  output logic [31:0] result_o,
  output logic        busy_o,
  output logic        valid_o
);
```

2.3 Deliverables

All files for this task need to be added in the folder `task-2`.

1. Edit the `README.md` file and describe which parts of your submission are complete to give your TA an overview.
2. Your RTL-model of the Cordic accelerator. The filename is `src/cordic.sv`
3. Any testvector you add yourself. Place them in the `test` folder.

Add all these files to the git repository. Make sure to commit your files and push them to your git repository on GitLab. Also, do not forget to create a tag and push it according to Section -1.6.

2.4 Hints

- Run `make lint` to statically analyze your SystemVerilog design for errors.
- Make use of SystemVerilog's `signed` keyword.
- Build a finite state machine for the accelerator and integrate the required arithmetic.
- Run the `make test` command to test your implementation against all testvectors in the `test` directory.
- Run the `make run TARGET=<target>` command to test your implementation against a specific testvector. Exchange `<target>` with the name of the testvector, omitting the file extensions.
- For debugging purposes, it is helpful to visualize the timing behavior of the signals. For this, you can use *gtkwave* by running `make view TARGET=<target>`.

2.5 Test Vectors

To test your circuit for correctness, you get test vector and the corresponding result vector. Use this data to verify the correctness SystemVerilog implementation. Note, this data format is important. The first column in the test vector corresponds to the angle (Q3.28 number format) and the second column controls whether the sine (0) or cosine (1) needs to be computed. The result vector contains the computed sine or cosine value (Q3.28 number format). Again, note that the given test vector is **not exhaustive** and you will need to add additional testvectors to test the full capability of your implementation.

Listing 2.2: Test Vector

```
057de2dd 0
057de2dd 1
2b265454 0
2b265454 1
45e6d667 0
45e6d667 1
51923683 0
51923683 1
f517c65d 0
f517c65d 1
d66da302 0
d66da302 1
bbb5d3fc 0
bbb5d3fc 1
b0af8d79 0
b0af8d79 1
00000000 0
```

Listing 2.3: Result Vector

```
056271a4
0f1115e5
06e22ce2
f18e707c
f0ef5367
fa9c67e5
f12d13c1
06055aee
f5eb112c
0c6c90f4
f7ba4f02
f24deb72
0e725c59
f91f8135
0f85ede6
03e085e8
```

3 Task 3: CPU Integration

The first goal is to implement a multiplier as a Register-Transfer-Level (RTL) model in the hardware description language SystemVerilog. Multiplication is a non-trivial operation. In fact, even RISC-V does not include multiplication in its base integer instruction set. A dedicated M extension provides multiplication in RISC-V. In this task, you need to implement a simple multiplier using the *shift-and-add algorithm* in a dedicated module.

In a second step, you need to integrate this multiplier in the MicroRISC-V CPU by adding multiplication instructions according to the RISC-V instruction set. Finally, you will implement instruction fusion as a performance optimization.

3.1 Shift-and-add Algorithm

The shift-and-add algorithm is a simple algorithm that allows implementing multiplication by using only left-shift and addition operations. For clarity we only consider 4-bit values in the example given below. Note that your implementation needs to operate on 32-bit values. Let A be binary `1011` and let B be binary `0110`. The multiplication can be visualized as follows:

```
      0 1 1 0  •  1 0 1 1   (B•A)
      -------
      0 1 1 0               (0)
    0 1 1 0                 (1)
  0 0 0 0                   (2)
0 1 1 0                     (3)
-------------
1 0 0 0 0 1 0               (result)
```

We process the bits of A from the least significant (rightmost) to the most significant (leftmost) position. Since the first bit is one, we copy value B to line `(0)`. The second bit is one, we copy value B to line `(1)` and shift it by 1 position to the left. The third bit is zero, hence we copy zeroes to line `(2)` and shift it by 2 positions to the left. Finally, due to bit four as one, a copy of B is shifted by 3 positions to the left. Algorithmically speaking, for bit index i we shift B (if the i-th bit is 1) or zeroes (if the i-th bit is 0) by i positions.

Once we are finished with shifting, we apply addition. Columnwise, we add up the values of rows `(0)`, `(1)`, `(2)`, and `(3)`. This yields the final result of our multiplication. Indeed, $6(0b110) \cdot 11(0b1011) = 66(0b1000010)$.

3.2 Specification

3.2.1 Multiplier

Your multiplier must implement the interface shown in Figure 3.1. clock_i and rst_i signals are input signals for every synchronous machine. a_i, b_i, and result_o relate to the multiplication operation. start_i, busy_o, and finish_o are status signals indicating the progress of the computation.

```
module multiplier (
  input logic         clk_i,
  input logic         rst_i,
  input logic         start_i,
  input logic [31:0]  a_i,
  input logic [31:0]  b_i,
  output logic        busy_o,
  output logic        finish_o,
  output logic [63:0] result_o
);
```

Figure 3.1: MUL top-level interface in SystemVerilog.

Your circuit must fulfill the following specification:

- Initially after the reset, the control signals start_i, busy_o and finished_o are 0.
- When start_i is set to 1, the computation starts. The shift-and-add algorithm is applied to 32-bit values a_i and b_i.
- Therefore, the circuit must satisfy the following protocol:
  1. Once start_i is set high and clk_i has some positive edge, let A be the value of signal a_i and B be the value of signal b_i
  2. After a finite amount of clock cycles let finish_o become high.
  3. If finish_o is high, the result_o output must hold the value $A \cdot B$.
- When the computation is finished the control output finished_o is set to 1 and the computed result can be read from result_o.
- Given two 32-bit input values, the result is a 64-bit value.
- Make sure that the reset signal rst_i is implemented as an asynchronous reset.
- You can assume that between start_i becoming high and finish_o becoming high, the values a_i and b_i do not change.

- You need to submit one SystemVerilog file with your implementation. The top-level module must be called "multiplier.sv".

The start_i signal should be reset to 0 at earliest one clock cycle after setting it to 1 and at latest when the finished_o signal changes from 0 to 1 (when the multiplication is done). Since, this is a trivial multiplication, testcases are not listed here. Be sure not to implement latches by following our coding style conventions.

3.2.2 MicroRISC-V Integration

MicroRISC-V is a single-cycle CPU that implements a subset of the RISC-V RV32I instruction set. The implementation is based on a classic CPU datapath rather than a state machine. Figure 3.2 shows the microarchitecture of the implemented CPU. Your task is to extend the CPU with the `MUL` and `MULHU` instructions of RISC-V's M extension.

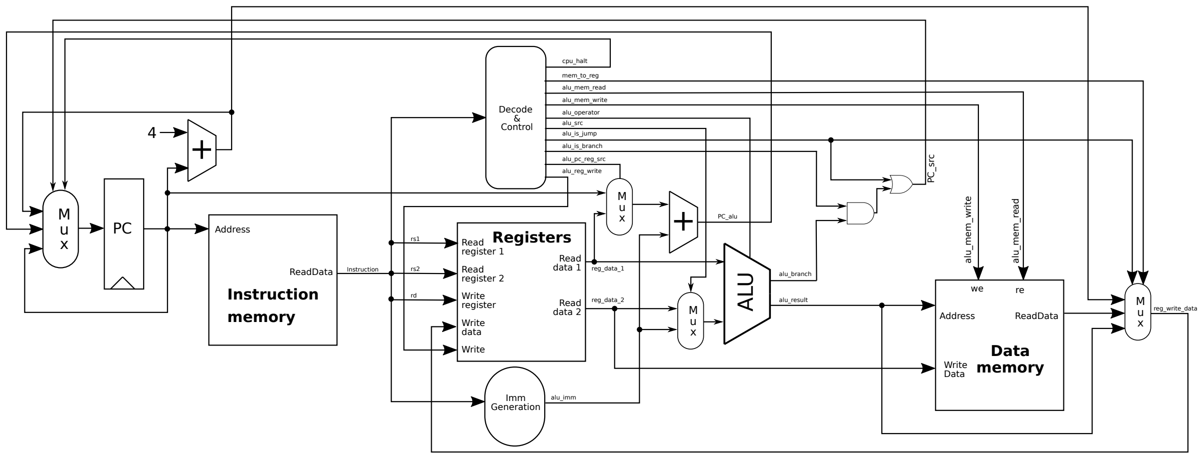

Figure 3.2: MicroRISC-V architecture.

Table 3.1: Encoding of instructions to be implemented.

31 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0	
0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	011	rd	0110011	MULHU

Integrate the multiplier to the given MicroRISC-V CPU by instantiating the multiplier module within the CPU. Extend the decoding and executing steps of the CPU for the multiplier. RISC-V defines two intructions for performing an unsigned 32-bit×32-bit multiplication:

Note that the multiplication operation is a multi-cycle operation. Thus, it requires to stall the CPU until the multiplication algorithm finishes and its result is available.

Make sure that your hardware design does not contain latches! Use the command `make synth` to synthesize your HDL code to hardware using Yosys. The resulting area log output contains information whether the design contains a latch. A latch is found if an element with `LATCH` in its name was created (e.g. `$_DLATCH_N_8`).

Given that the possible range of results for the multiplication exceeds the 32 bits that can be stored in a register it is necessary to have two multiplication operations:
`MUL rd, rs1, rs2`. Perform a 32-bit×32-bit multiplication between `rs1` and `rs2` and store the **lower** 32-bit of the result in the register `rd`.
`MULHU rd, rs1, rs2`. Perform a 32-bit×32-bit multiplication between `rs1` and `rs2` and store the **upper** 32-bit of the result in the register `rd`.

3.2.3 Instruction Fusion

To compute a 32-bit×32-bit multiplication and retrieve the final 64-bit result with the available multiplication instructions one ends up with an instruction sequence similar to the one below:

```
MULHU x3, x1, x2
MUL   x4, x1, x2
```

Figure 3.3: Typical multiplication sequence in RISC-V assembly.

Obviously, restarting the multiplier and executing the algorithm again with the same input data has a negative impact on performance. Thus, the RISC-V ISA manual recommends to fuse both instructions and perform the multiplication algorithm only once.

> "If both the high and low bits of the same product are required, then the recommended code sequence is: MULHU rdh, rs1, rs2; MUL rdl, rs1, rs2 (source register specifiers must be in same order and rdh cannot be the same as rs1 or rs2). Microarchitectures can then fuse these into a single multiply operation instead of performing two separate multiplies."
> —RISC-V ISA specification, "M" Standard Extension

It is your task to adapt the processor to detect a multiplication as described above and only invoke the multiplication once. The second multiplication instruction reuses the result of the previous multiplication. Only the consecutive execution of `MULHU` and `MUL` in this order must be fused.

3.3 Deliverables

All files must be submitted in folder `task-3` of your repository.

1. Edit the `README.md` file and describe which parts of your submission are complete to give your TA an overview.
2. Submit your SystemVerilog module for multiplication in `ips/multiplier.sv`.
3. Submit your modified MicroRISC-V implementation including all files of the upstream repository. This implementation is supposed to feature `MUL` and `MULHU` as well as implement instruction fusion.

Add all these files to the git repository. Make sure to commit your files and push them to your git repository on GitLab. Also, do not forget to create a tag and push it according to Section -1.6.

3.4 Hints

- Run `make lint` to statically analyze your SystemVerilog design for errors.
- Run `make multiplier` to build the multiplier and its isolated testbench, run `make run_multiplier` to run it, and `make view_multiplier` to view its waveforms.
- Run `make run TARGET=<program-name>` to generate `_sim/riscv_core.vvp` simulating the CPU and executing the `<program-name>.asm` testcase. Look into the `testcases` folder for different test programs.
- Run `make view TARGET=<program-name>` to simulate your CPU and view the signal trace with GTKWave.
- Run `make sim TARGET=<program-name>` to execute the program on the ISA simulator to observe the expected behavior.
- Run `make test` to compare the output of the ISA simulator and the hardware implementation against the expected output values.
- `riscvasm.py` does not understand `MUL` instructions. Use the `-e` flag as follows: `riscvasm.py -e mul_extension.py testcase.asm > testcase.hex`. The Makefile does this automatically for you.
- For testing: 32-bit integers cannot be loaded in RISC-V with a single instruction. Use a `LUI`+`ADDI` sequence as explained on Stackoverflow. The `LI` pseudoinstruction is not supported by `riscvasm.py`.
- When not using the provided virtual machine, be sure to use Icarus Verilog 11. Older versions contain bugs where a design might freeze in simulation.
- Pay attention to the preconditions for instruction fusion.

4 Task 4: XGCD in RISC-V

The goal of this task is to get comfortable with the lowest abstraction level of software: the Instruction Set Architecture (ISA) by writing software in assembly language. In this task, we use the Extended Euclidean Algorithm algorithm in RISC-V. Because the algorithm (among others) determines the Greatest Common Divisor, the algorithm is commonly abbreviated as XGCD. First, you implement the XGCD algorithm in C. Second, you transform the C code into an assembly-like representation. Finally, you implement this algorithm in pure assembly and execute it on the RISC-V simulator.

4.1 Extended Euclidean Algorithm

The xgcd algorithm takes two integers (a and b) and returns three numbers (x, y, and gcd) satisfying $\mathsf{gcd} = \mathsf{x} \cdot \mathsf{a} + \mathsf{y} \cdot \mathsf{b}$. gcd is a positive integer dividing a and b. For example, $3 = 28 \cdot 36 + (-5) \cdot 201$ gives 3 as greatest common divisor of the numbers 36 and 201. In our setting, we are going to restrict a and b to the domain $[1, 2^{31} - 1]$. You can read up on the significance of this algorithm on Wikipedia or "The Art of Computer Programming", Volume 2, Chapter 4. Algorithm 2 illustrates the pseudo code for the XGCD algorithm.

4.2 Specification

Our goal is to implement the function `xgcd` and some wrapper around it in RISC-V assembly language. To get started, you implement the XGCD algorithm in the C programming language (`xgcd.c`). Afterward, you are going to transform it in C in such a way that each line of C code (except for function entries and returns) can be mapped 1:1 to assembly instruction. Subsequently, the task is to convert the implementation to assembly. The following description of `xgcd.c` gives a short overview of the used functions.

`main` The main function allocates an integer variable `size` for the number of elements and 5 arrays for the values to be processed on the stack. Then it calls `input`, `xgcd`, and `output` in succession.

`input` first reads the number of elements being processed from stdin. Then it alternately reads the a and b values and stores it in the given arrays.

`output` prints all five processed values to stdout.

Algorithm 2 Extended Euclidean Algorithm

```
procedure XGCD(size, a, b, x, y, gcd)
    declare $x\_prev[2]$
    declare $y\_prev[2]$
    declare $r, a\_tmp, b\_tmp$
    declare $q, x\_next, y\_next$
    for $i \leftarrow 0$ to $size - 1$ do
        $x\_prev[0] \leftarrow 1$
        $x\_prev[1] \leftarrow 0$
        $y\_prev[0] \leftarrow 0$
        $y\_prev[1] \leftarrow 1$
        $r \leftarrow 1$
        $a\_tmp \leftarrow a[i]$
        $b\_tmp \leftarrow b[i]$
        while $r \neq 0$ do
            $q \leftarrow a\_tmp/b\_tmp$
            $x\_next \leftarrow x\_prev[0] - q \cdot x\_prev[1]$
            $y\_next \leftarrow y\_prev[0] - q \cdot y\_prev[1]$
            $r \leftarrow a\_tmp - q \cdot b\_tmp$
            if $r \neq 0$ then
                $a\_tmp \leftarrow b\_tmp$
                $b\_tmp \leftarrow r$
                $x\_prev[0] \leftarrow x\_prev[1]$
                $x\_prev[1] \leftarrow x\_next$
                $y\_prev[0] \leftarrow y\_prev[1]$
                $y\_prev[1] \leftarrow y\_next$
            end if
        end while
        $x[i] \leftarrow x\_prev[1]$
        $y[i] \leftarrow y\_prev[1]$
        $gcd[i] \leftarrow b\_tmp$
    end for
end procedure
```

`xgcd` computes the Extended Euclidean Algorithm for each pair of `a` and `b`. It iterates over all array elements and stores the results `x`, `y`, and `gcd` in the arrays provided via the parameters.

All three implementations read input via `stdin` and write output via `stdout` in the same format. Each line consists of a 8-digit hexadecimal signed number. The first line denotes the number of elements. Then, the files consist of alternating lines for `a` and `b` for the number of element times. Output files consist of 5-tuples with values of `a`, `x`, `b`, `y`, and gcd respectively per line. The maximum number of input pairs is limited to 10.

You can compile all three implementations using the provided Makefile with `make`. The executables are written to the folder `_sim`. You can either supply input files manually like `_sim/xgcd.elf < test/input_01.testvec` or run `make test`, which provides a test suite.

`xgcd.c` In this file, the two functions `main` and `xgcd` are not implemented. Use this file to implement the XGCD algorithm in the C programming language. The pseudo code for the XGCD algorithm is illustrated in Algorithm 2, which you should use for your implementation of the `xgcd` function.

`xgcd_transformed.c` In this file, the functions `main` and `xgcd` are not implemented. It is your task to transform the previously implemented functions into an assembly-like representation. Precisely, transform these functions in such a way that each line in `xgcd_transformed.c` corresponds to precisely one instruction in `xgcd.asm`. This holds for all lines except for the function entries and exits, which lead to corresponding prologues and epilogues in assembly. Note, the function `xgcd` must not have any parameters. Apply the RISC-V calling convention for passing parameters via the global registers to subroutines. For the implementation, observe the following rules:

- The functions `input` and `output` are already implemented. Use them and all other functions according to the RISC-V calling convention.
- Only use the registers declared in the top of the file. Use them to compute intermediate values and to pass arguments to other functions (just like you would do with registers in RISC-V). You must follow the RISC-V calling convention for argument and return value passing. Local variables are not allowed, except for using as a stack slot for the callee saved registers `s1-s11`.
- The function `main` allocates all arrays and local variables on the stack.
- Replace all if/else statements and loops with single if/goto statements in order to achieve the requirement that each line must map 1:1 to an assembly instruction (except for function entry and return). The input and output functions are given as examples.
- All array accesses must be resolved to dereferenced pointer accesses.

Keep in mind, that this file should ease your C to assembly language conversion.

xgcd.asm This file is supposed to contain your assembly implementation. The same functions are missing and are required to be implemented. You can easily convert the transformed C implementation to assembly. The XGCD algorithm requires a multiplication to compute the remainder.

In this task, the RISC-V instruction set was extended with the *M*-extension, thus providing support for multiplication, division, and remainder computation. Note, due to the value range of the remainder computation, you only need a single `MUL` instruction rather than a full pair of `MUL/MULSU`. You can explore the python file `m_extension.py` for the added instructions. Refer to the RISC-V ISA specification for the detailed semantics of the instructions of the *M*-extension.

In addition to that, you must maintain the stack for storing the return addresses, local variables, and spilled registers. Obey the following rules:

- The functions `input` and `output` are already implemented. Use them according to the RISC-V calling convention.
- All function calls must follow the RISC-V calling convention.
- Registers shall only be used for their intended application binary interface (ABI) purpose.
- The registers `ra` and `sp` must contain their designated value (i.e. return address and stack pointer) all the time. Even if you don't use them in some functions.
- If you don't have enough temporary registers, you can use the callee saved registers `s1-s11`. Note, these must be spilled on the stack before you are allowed using them.
- Maintain a proper function prologue and epilogue.

In this task, you use `riscvasm.py` for assembling the source code. This assembler has a a limited set of supported instructions. The following RISC-V instructions are supported and can be used:

- **Arithmetic**: `ADD`, `ADDI`, `SUB`, `AND`, `OR`, `XOR`, `SRA`, `SRL`, `SLL`
- **M-Extension**: `MUL`, `MULH`, `MULHSU`, `MULHU`, `DIV`, `DIVU` `REM`, `REMU`
- **Memory Access:** `LW`, `SW`
- **Conditional Branches:** `BEQ`, `BNE`, `BLT`, `BGE`
- **Jumps:** `JAL`, `JALR`
- **Miscellaneous:** `LUI`, `EBREAK`

4.3 Deliverables

All files must be submitted in folder `task-4` of your repository. All files of the upstream repository must be included!

1. Edit the `README.md` file and describe which parts of your submission are complete to give your TA an overview.
2. Modify `xgcd.c` to provide your C implementation of the functions `main` and `xgcd`.
3. Modify `xgcd_transformed.c` to provide your transformed implementation of the functions `main` and `xgcd`.
4. Modify `xgcd.asm` to provide your assembly implementation of the functions `main` and `xgcd`.

Add all these files to the git repository. Make sure to commit your files and push them to your git repository on GitLab. Also, do not forget to create a tag and push it according to Section -1.6.

4.4 Hints

- Follow the transformation steps taught in the lecture or the tutorial video. Make one step after another and study the RISC-V instruction set.
- Don't forget to resolve then-blocks of conditionals. Remove curly parentheses and use the pattern `if (cond) goto label_after_then_block;`
- If a register, e.g., `t0`, contains a memory address, use the pattern `t1 = *(int*)t0` to emulate a load into register `t1`.
- Pseudoregisters have the type `size_t`. This ensures that they are large enough to be able to store pointers in them. Use casts to switch between C integers and C pointer values when needed.

5 Task 5: QUIB

The QUIC protocol, used in HTTP/3, is a reliable data transmission protocol that is intended to replace TCP. It is connection-based like TCP, but allows multiple independent byte streams within a single connection, which are loss-controlled independently. Losing data for a particular stream only delays that particular stream. QUIC also integrates cryptographic key agreement with the transport layer handshake, allows connections to “float” between remote hosts if the host moves to a different network, integrates latency calculations into the protocol headers, allows ahead-of-time encrypted data to be sent before the handshake even completes, and many more.

No, stop. Don’t run away. We won’t be asking you implement most of these features. Instead, you will implement a heavily feature-reduced version of QUIC that focuses on the basics of stream transmission and acknowledgement We will call this protocol QUIB. It is described below.

5.1 QUIB basics

QUIB is a connection-based protocol. It establishes a connection between two peers - a “client” and a “server”. The server waits for incoming connections on a particular port. The client connects to a chosen server. You will implement a QUIB client in this assignment. The QUIB protocol is detailed below. Your tasks are summarized in Section 5.4.

5.1.1 QUIB Streams

A “stream” is a separate data exchange context inside a single connection. Multiple streams in the same connection are independent. A stream can either be “unidirectional”, meaning only the stream’s creator can send data on it, or “bidirectional”, meaning that both peers can send data on it. Even if a stream is bidirectional, the sending and receiving parts are independent. Data sent by the client is read by the server, and data sent by the server is read by the client.

Each stream direction transports a long stream of bytes. Streams have no concept of “messages”. This means that sending 0x01 (1 byte), then sending 0x02 (1 byte), has the same effect as sending 0x0102 (2 bytes).

Either side of the stream can terminate it. For bidirectional streams, the sending and receiving parts are terminated independently from each other.

5.1.2 QUIB data transmission

QUIB transmits frames of information. One or more frames are combined to form a packet. Each packet is transmitted as a UDP datagram. Connections are between a client and a server, similar to TCP. They are identified by a pair of connection IDs, one chosen by the client and used by the server, and one chosen by the server and used by the client. Each packet sent on a connection is numbered, and receipt of packets is acknowledged by each peer. Packet numbers are maintained separately by client and server. Acknowledging a packet confirms receipt of all contained frames. If no acknowledgement is received, the frames contained in the packet will be resent in a new packet.

Packets are laid out as follows. The number in parenthesis is the number of bits for the component. There is no padding between fields. The maximum size of a packet is 3 MB (3,145,728 bytes).

```
Packet {
  packet number (8..32)
  destination connection id (160)
  source connection id (0 or 160)
  payload (8..)
}
```

The packet number is encoded as a variable-length integer (see Section 5.3.1). The special value 0b11 (0x3) for its length marker is used to denote an “initial packet”, which is part of a QUIB handshake. Initial frames have a 6-bit packet number. (You treat the 0b11 length marker as if it were 0b00.) The source connection id is only present for initial packets. The payload contains a sequence of one or more QUIB frames, which are simply concatenated to each other.

5.1.3 The QUIB handshake

To establish a QUIB connection, a packet with packet number length marker 0b11 (an “initial packet”) and all-zeros destination connection id is sent to the server. A random, unused, non-zero, client connection id should be chosen for this connection, and sent as the source connection id. The packet number for the connection starts at zero, and the first attempt at sending the initial packet should use packet number zero. If the packet is retried, the same client connection id, but a new packet number, should be used. At least one QUIB frame must be included in the payload. If any data to be sent is already available, this data can be in the initial packet’s payload. Otherwise, a single PING frame should be used.

When the server receives this initial packet, it responds with an initial packet of its own. The destination connection id is set to the client connection id chosen by the client. The source connection id is set to the server connection id, arbitrarily chosen by the server, and must not be all-zeros. The payload consists of one or more QUIB frames, the first of which is an ACK frame acknowledging receipt of the corresponding client initial packet.

If a peer receives more than one initial packet for a particular connection, any future initial packets on that connection must be ignored. Their receipt should not be acknowledged. Any data contained should not be considered.

5.1.4 Non-initial packets

The client may begin sending non-initial packets as soon as it has received the server's initial packet. The server may begin sending non-initial packets as soon as it has sent its initial packet. (It does not need to wait for the initial packet to be ACK'd.)

Non-initial packets have similar structure to the initial packets exchanged in the handshake, but do not contain the special 0b11 length marker. This means that non-initial packets' packet number can use the full 30-bit variable-length integer range. Each packet sent by a peer must have a unique packet number. (Even if a packet is lost, and data contained is re-sent, this is done in a new packet, with a new packet number.) Packet numbers should be used one after the other, with no numbers being skipped. This allows ACK frames (see Section 5.2.2) to remain compact. The destination connection id is set to the connection id provided by the peer in its initial packet. No source connection id is included. The payload starts immediately following the destination connection id. The payload consists of one or more QUIB frames. Packets must not have an empty QUIB payload.

5.2 QUIB frames

A QUIB packet's payload consists of one or more frames. The frames are simply concatenated, and processed in order. Each QUIB frame starts with a one-byte type field. The remainder of the frame depends on the type. Each type is described below.

5.2.1 PING (type 0x01)

A single-byte contentless frame. Carries no meaning, but must be acknowledged. This can be used to force the peer to send an acknowledgment and ensure the connection is alive. See Section 5.4.10 for details on how to handle connection timeout.

5.2.2 ACK (type 0x02)

A frame acknowledging the receipt of packets from the peer. It has the following layout.

```
ACK {
  type (8) = 0x02
  largestReceived (8..32)
  firstRange (8..32)
  extraRangeCount (8..32)
  ackRanges (0..)
}
```

largestReceived is the packet number of the highest packet being acknowledged, in variable integer form (cf. Section 5.3.1). firstRange is the number of contiguous packet numbers, below the largest, being acknowledged. If you are acknowledging the largest, and the one below it, this number would be one. If you are acknowledging just the largest, it is zero.

extraRangeCount is the number of additional acknowledged ranges following, in variable integer form. It can be zero. Each range consists of two variable-length integers. The first expresses the "gap", i.e., the number of packet numbers not received, below the previous range, minus one. The second expresses the number of packet numbers in the range, minus one.

For example, assume you are acknowledging packet numbers 44, 43, 42, 40, 39, 37, 34, and 33. In other words, you are acknowledging packets 44 through 42, 40 through 39, 37 through 37, and 34 through 33. This would be translated into an ACK frame as follows:

```
ACK {
 type (8) = 0x02
 largestReceived (8) = 44
 firstRange (8) = 2 /* = 44 - 42 */
 extraRangeCount (8) = 3
 ackRange (16) {
   gap (8) = 0 /* = (41 - 40) - 1 */
   range (8) = 1 /* = 40 - 39 */
 }
 ackRange (16) {
   gap (8) = 0 /* = (38 - 37) - 1 */
   range (8) = 0 /* = 37 - 37 */
 }
 ackRange (16) {
   gap (8) = 1 /* = (36 - 34) - 1 */
   range (8) = 1 /* = 34 - 33 */
 }
}
```

When constructing an ACK frame, all received packets for which the peer has not previously acknowledged an acknowledgement should be included. Once the peer has acknowledged an acknowledgement for a given range, that range should no longer be included in future ACK frames.

If a packet contains only ACK and CONNECTION_CLOSE frames, it is called a "non-ack-eliciting" packet. If the only packets you need to acknowledge are "non-ack-eliciting" packets, you should not acknowledge them unless there are other frames you'd like to send. (Otherwise, each acknowledgement would prompt another acknowledgement by the peer, and the connection would never become idle.)

5.2.3 STREAM (type 0x08 through 0x0f)

The type byte of STREAM frames is of form 0b00001XXX. Each of the three flexible bits carries additional meaning. If the 0x04 bit is set, a byte offset is present in the frame. If the bit is unset, the offset is not specified, and implicitly zero. If the 0x02 bit is set, an explicit length field is present in the frame. If the bit is unset, this is the last frame of the packet, and its data spans to the end of the packet. If the 0x01 bit is set, the stream's data ends with this frame. In other words, it is set if total stream size = frame offset + frame size.

A stream frame is laid out as follows:

```
STREAM {
  type (8) = 0x08..0x0f
  streamID (8..32)
  offset (0 or 8..32)
  length (0 or 8..32)
  data (0..)
}
```

The stream identifier is always present. It is a variable-length integer. See Section 5.3.2 for information on stream identifiers. The offset is only present if the 0x04 bit is set in the type field. If it is present, it is a variable-length integer expressing the index of the first byte included, minus one. The length is only present if the 0x02 bit is set in the type field. If it is present, it is a variable-length integer expressing the length of the data. The remainder of the frame consists of the actual data being sent. If an explicit length is present, the data is of that length. If no explicit length is present, this is the last frame of the packet, and the data is the remainder of the packet.

If no stream with the specified identifier is known, and the stream identifier is for a stream initiated by the other peer (see Section 5.3.2), the stream is implicitly opened.

5.2.4 MAX_STREAM_DATA (type 0x11)

This frame increases the flow control limit for a given stream. It specifies the maximum index (in bytes) which may be sent by the peer, plus one. MAX_STREAM_DATA can only increase the flow control limit; it can never reduce it. When a stream is created, its implicit flow control limit is 1 MB (1,048,576 bytes). Data at positions 0 through 1048575 may be sent.

```
MAX_STREAM_DATA {
  type (8) = 0x11
  streamID (8..32)
  maxSize (8..32)
}
```

5.2.5 RESET_STREAM (0x04)

This frame abruptly terminates the sending part of a stream. Any data previously sent on the stream will no longer be re-sent, and the receiver may discard any data it previously received on the stream. Any attempts to read from the stream should fail.

```
RESET_STREAM {
  type (8) = 0x04
  streamID (8..32)
}
```

5.2.6 STOP_SENDING (0x05)

This frame abruptly terminates the receiving part of a stream. It indicates that the application is no longer interested in receiving data on this stream. No more data should be sent on this stream. This includes previously-enqueued data that has not yet been acknowledged.

```
STOP_SENDING {
  type (8) = 0x05
  streamID (8..32)
}
```

5.2.7 CONNECTION_CLOSE (0x1c)

This frame indicates that the connection is being closed. Any incomplete receiving streams on the connection should be treated as having been reset. If a packet containing this frame is received, no further packets must be sent on the connection. CONNECTION_CLOSE may not be sent alongside any non-ACK frames. Therefore, a packet containing CONNECTION_CLOSE will never be acknowledged. If the connection is closed by the application, you should wait until all stream data for complete streams has been acknowledged before sending CONNECTION_CLOSE.

```
CONNECTION_CLOSE {
  type (8) = 0x1c
  errorCode (8)
}
```

If the application requests connection closure, the error code is NO_ERROR (0x00). If the protocol dictates connection closure due to a misbehaving peer (cf. Section 5.3.3), the error code is set accordingly.

5.3 Additional Information

5.3.1 Variable-Length Integer Encoding

Variable-length integer encoding is designed to represent small integers in as little space as possible, while still allowing a large range of integers to be represented. When reading an variable-length integer in the context of QUIB, look at the two most-significant bits of the first byte. If these bits are 0b00, this is a one-byte integer, contained in the six low bits of the byte you are looking at. If these bits are 0b01, this is a two-byte integer (14 usable bits). If they are 0b10, this is a four-byte integer (30 usable bits). The value 0b11 has special meaning in the context of packet numbers, and is otherwise unused in QUIB.

Here are some examples. The integer 27 (`0b00011011`) is encoded as one byte: `00011011`. The integer 91 (`0b01011011`) is encoded as two bytes: `01000000 01011011`. The integer 16,677 (`0b01000001 00100101`) is encoded as four bytes: `10000000 00000000 01000001 00100101`.

5.3.2 Stream Identifiers

Stream identifiers are numbers between 0 and $2^{30} - 1$.

The least significant bit of the stream identifier identifies which peer may open a given stream. When the client opens a stream, it uses an even stream identifier (LSB 0). When the server opens a stream, it uses an odd stream identifier (LSB 1).

The second least significant bit of the stream identifier identifies whether the non-opening peer may send data on this stream ("bidirectional", byte = 0) or not ("unidirectional", byte = 1). For example, the stream with identifier 21 (0b00010101) is a server-initiated bidirectional stream. The stream with identifier 26 (0b00011010) is a client-initiated unidirectional stream.

Streams are not explicitly created. If a stream identifier is referenced for the first time in a packet that uses stream identifiers, this implicitly creates the stream.

5.3.3 Error Handling

If you receive invalid data from a peer, and can identify the data as belonging to a connection, you should close that connection. You will do this even if data is pending for the connection, or there are outstanding acknowledgments. Depending on the type of error, you should specify an error code. If you cannot recognize a frame, use error code FRAME_ENCODING_ERROR (0x07). If you receive data for a client-initiated stream identifier which you did not initiate, use error code STREAM_STATE_ERROR (0x05). If you receive stream data for an offset beyond the previously-established stream size (based on a previous STREAM frame with 0x01 set), use error code FINAL_SIZE_ERROR (0x06). For any other erroneous behavior, you can always use the catch-all error code PROTOCOL_VIOLATION (0x0a).

5.4 Your tasks

You will implement a QUIB client as a library, which programs can use to communicate. Start by looking at the header file `quib.h`. It contains the functions you will need to implement. These functions will be called by the program using your library. The behavior they should have is described below.

To simplify your task, time will be measured in discrete "ticks". The task framework will call the function `quib_tick` regularly. You will implement this function to receive incoming packets, process them, and generate new packets to send. You should not receive or send packets outside of this function. To send or receive packets (as UDP datagrams), use `udp_open`, `udp_close`, `udp_send_datagram`, and `udp_receive_datagram`. These functions are implemented by us. You can (and should) simply use them. (Do not manually send UDP datagrams using system functions. This will not work with our test system.)

5.4.1 quib_connect

This function is called when the application wants to create a connection. You should open a udp socket for sending to/receiving from the destination (`udp_open`), and initiate your internal connection state. You should not send any data in this function. Wait until the next `quib_tick` to send data. Once you have initialized the connection state, set `(*pcHandle)` to a handle to it, and return `NO_ERROR`. This handle should remain valid until it is discarded by the application.

5.4.2 quib_is_connected

This function checks whether the connection has been successfully established. It should not change the connection state. It returns true if, and only if, the connection handshake has succeeded (you have received an initial packet from the server), and the connection is still open (it has not been closed).

5.4.3 quib_close_connection

This function closes the connection. The handle remains valid, and might still be used by the application. Any future operations on the handle should do nothing, and return `ERR_CONNECTION_CLOSED` as documented. If the connection is already closed or timed out, this function does nothing.

5.4.4 quib_discard_connection

If nullptr is passed, this function does nothing. Otherwise, this function discards the connection handle. You can assume that the connection handle will no longer be used by the application. If the connection is still open, it should be closed. Keep in mind you might need to keep some connection state stored until you have done so (and free it afterwards!) If there are any stream handles associated with the connection, they remain valid until they are discarded.

5.4.5 quib_new_stream

This function creates a new stream in the specified connection. Once you have initialized the stream state, set `(*psHandle)` to a handle to it, and return `NO_ERROR`. This handle should remain valid until it is discarded by the application. If the connection is closed, return `ERR_CONNECTION_CLOSED`. If it has timed out, return `ERR_CONNECTION_TIMEOUT`. The state of `(*psHandle)` is unspecified in these cases.

5.4.6 quib_accept_stream

This function should check if the server has initiated a new stream in this connection. If this has happened, set `(*psHandle)` to a handle to the stream, and return `NO_ERROR`. If this has not happened, set `(*psHandle)` to `nullptr`, and return `NO_ERROR`. If the connection is closed, return `ERR_CONNECTION_CLOSED`. If it has timed out, return `ERR_CONNECTION_TIMEOUT`. The state of `(*psHandle)` is unspecified in these cases.

5.4.7 quib_discard_stream

If nullptr is passed, this function does nothing. Otherwise, this function discards the stream handle. You can assume that this stream handle will no longer be used by the application. If the stream's sending part is open, it is reset. If the stream's receiving part is open, it is stopped.

5.4.8 quib_send_data

This function queues the specified data for sending on this stream. You should make a copy of the passed data. The passed pointer is not guaranteed to remain valid after you return from the function. Once you have done so, return `NO_ERROR`. If `isFinal` is `true`, these are the final bytes that will be sent on the stream. (You should indicate this when sending them, cf. Section 5.2.3.)

Keep in mind that this function should not send any actual packets. Only `quib_tick` may send or receive packets.

If you do not have sufficient storage to store the passed data, store none of it, and return `ERR_BUFFER_EXCEEDED`. Your implementation must be able to store at least 5 MB (5,242,880 bytes) of unsent data per write-capable stream. Additionally, data that you have successfully sent, and which has been acknowledged by the peer, does not count against this limit. If the stream is a unidirectional stream initiated by the server (which you cannot write on), return `ERR_READ_ONLY_STREAM`. If the stream's writing part has been closed already, such as because a previous call specified isFinal, return `ERR_STREAM_WRITE_CLOSED`. If the underlying connection is closed, return `ERR_CONNECTION_CLOSED`. If it has timed out, return `ERR_CONNECTION_TIMEOUT`.

5.4.9 quib_receive_data

This function tries to receive data on the specified stream. You should return as much as data as you have received, but never more than the specified size. Store it in the buffer pointed to by `pBuf`. Set the amount of data you have actually returned to `(*pNumReceived)`. This may be zero. If this is the last read that will produce data on the stream – i.e., the stream was closed regularly (by setting 0x01 on the STREAM frame type), and this read consumes all remaining data – set `(*pIsFinal)` to `true`. In this case, future read operations on the stream will produce `(*pNumReceived) == 0` and `(*pIsFinal) == true`. Otherwise – if the stream is still open, or this read operation does not consume all remaining bytes, or both – you should set `(*pIsFinal)` to `false`.

If the stream is a unidirectional stream initiated by the client (which you cannot read on), return `ERR_WRITE_ONLY_STREAM`. If the stream has been reset by the sender, or closed for reading by the application, return `ERR_STREAM_RESET`. In all of these error cases, the state of `(*pNumReceived)` and `(*pIsFinal)` are unspecified.

Your streams should have sufficient storage to store at least 5 MB (5,242,880 bytes) of data which the application has not yet received per stream. You should use flow control frames (MAX_STREAM_DATA, cf. Section 5.2.4) to communicate the remaining limit to the peer.

5.4.10 Packet Loss & Connection Timeout

For purposes of this assignment, you should consider a packet lost if it has not been acknowledged within two ticks of being sent. At this point, you should assume that the server did not receive any frames contained, and generate new frames containing the data. Note that you should not always re-send identical frames to the ones that were lost. For example, if you sent a frame containing 100 bytes of data on a stream, and that frame is lost, but in the meantime you have another 100 bytes to send on that stream, you should send a new frame containing all 200 bytes, not re-send the original 100-byte frame. In other words, you are re-sending the information that was lost, not the frames.

You should make sure to send at least one ack-eliciting packet (cf. Section 5.2.2) per two ticks. If this is the second tick, and you have no other frames to send, send a PING frame (Section 5.2.1) to ensure you receive acknowledgments from the peer. If you do not receive any packets from the peer for six ticks (despite sending ack-eliciting packets every other tick), you should assume that the peer has crashed, or is otherwise refusing to communicate with you. This is called "connection timeout". You should never send data on a connection that has timed out. Operations on such a connection will fail with `ERR_CONNECTION_TIMEOUT`. Connection timeout may also occur during the initial handshake. Even if you receive data from the peer after the connection has been timed out, you should not "revive" the connection.

5.5 Modalities

5.5.1 How to run & Test Cases

A complete test suite is included. It is the same test suite that will be used by us as the basis to evaluate your submission. There are no additional private tests.

However, note that we will also review your actual source code submission. Please do not design specifically only to pass the test cases.

Your submission should run without crashing or leaking memory. It will be run in Valgrind, which is a memory leak testing tool. You must not take steps to evade memory leak detection. Instead, you should fix any memory leaks in your code.

You can run the test suite using `make test`. This runs your submission in Valgrind for you. If you encounter any issues with the test framework itself, please raise them on Discord or with your TA.

5.5.2 Deliverables

All files must be submitted in folder `task-5` of your repository. You may also add additional files with ending `.cpp` or `.h` to this directory, and they will be processed by the test system. Do not add files with a different file extension, and do not add files to or modify files in sub-directories of `task-5`. Create a `README.md` file and describe which parts of your submission are (in-)complete to give your TA an overview.

Add all these files to the `git` repository. Make sure to commit your files and push them to your `git` repository on GitLab. Also, do not forget to create a tag and push it according to Section -1.6.

Errata

This section lists releases and changes of this file.

2024-10-09 Initial release.