Computer Organization and Networks Practicals 2022/23

November 7, 2022

Contents

0 Introduction

This document describes the tasks of the course "Computer Organization and Networks Practicals" for the winter term 2022/23. 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 the TCP/IP stack which defines our networks today.

0.1 Assignment Sheet

The assignment sheet that you are reading right now is provided in your repository. To ensure your version of the document is up to date, fetch updates from the repository using the following command:

```
git pull
```

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

0.2 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 to handle question hours and task interviews online. 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.
- #con is a generic channel for all CON participants to ask questions in textual form. Be kind to other participants and be aware, you are not allowed to post solutions to exercises. It serves as a place of discourse between students. TAs might attend but do not have to answer questions here. Have your repository number available when asking a tutor so he directly can look into your code.

- `#con-ta` is a prefix used for audio-only channels, one per TA. During their respective question hour, you can ask questions here and your TA will answer them.

0.3 Tutorial videos

Date	Content	Video
2022-10-06	Getting started	on seafile
2022-10-06	Task 1.a	on seafile
2022-10-06	SystemVerilog	on seafile
2022-10-06	Task 1.b	on seafile
2022-11-05	Task 2.a	on seafile
2022-11-05	Task 2.b	on seafile
2022-12-03	Task 3.a	
2022-12-03	Task 3.b	

Table 0.1: Tutorial session videos.

Tutorial videos are prerecorded videos (c.f. Table 0.1) to be published on the day of the deadline of the previous exercise. and will be shared on `#con`. The main goal is to introduce you to the next task and show you the required toolchain.

0.4 Toolchain

The toolchain is introduced in the tutorial videos (Section 0.3), 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 (→ sv2v) to convert SystemVerilog to Verilog
- Yosys for synthesis (→ Yosys Open SYnthesis Suite)
- Icarus Verilog (iverilog) 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 `Makefile`s 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 22.04:

`https://seafile.iaik.tugraz.at/f/4af9802977b24525b447/`

The user for this VM is `con` and the password `con2022`.

0.5 Question hours

	Monday	Tuesday	Wednesday
09:00	Group 1 Sebastian	Group 5 Patrick S.	Group 9 Markus
10:00	Group 2 Patrick K.	Group 6 Daniel	Group 10 Alexander
11:00	Group 3 Sarah	Group 7 Nives	Group 11 Felix
13:00	Group 4 Oliver	Group 8 Constantin	

Table 0.2: TA weekly question hour times.

Question hours are specific to your TA. They take place every week and you can look up your TA's timeslot in Table 0.2. In the #con-ta channel of your TA, you can ask questions about the tasks during the one hour question time. Note, this is an audio-only channel without messaging support.

0.6 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.a is submission-task-1a.
- 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 0.3. 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.

Task	Deadline	Max. points
Task 1.a Multiplier	Fr, 2022-10-28 23:59	max. 15 points
Task 1.b GCD Accelerator	Fr, 2022-11-04 23:59	max. 20 points
Task 2.a Peripheral Integration and Interrupts	Fr, 2022-11-25 23:59	max. 15 points
Task 2.b Binary Insertion Sort	Fr, 2022-12-02 23:59	max. 15 points
Task 3.a Firewall	Fr, 2023-01-13 23:59	max. 15 points
Task 3.b Internet Radio	Fr, 2023-01-20 23:59	max. 20 points

Table 0.3: Task submission deadlines and maximum achievable points.

0.7 Task Interviews

There are going to be two task interviews (for three tasks each). The first will cover tasks 1a, 1b and 2a. The second will cover tasks 2b, 3a and 3b. 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. For your task interview, join channel `#con-waiting-room` on Discord ten minutes before the interview. This is also the appropriate place to test your microphone. Your TA is going to pick you up at your designated time slot. Then both will join the `#con-ta` channel of your TA. In `#con-ta`, the task interview will be held.

The goal of the interviews is to verify you did the implementation yourself, understood the topic and collect feedback on both sides.

0.8 Grading

The maximum points per task are listed in Table 0.3. Depending on your achieved points, you will get a grade according to the following scheme.

$$
\begin{array}{rll}
0\text{–}50 & \text{Points} & \rightarrow & \text{Nicht genügend (5)} \\
51\text{–}62 & \text{Points} & \rightarrow & \text{Genügend (4)} \\
63\text{–}75 & \text{Points} & \rightarrow & \text{Befriedigend (3)} \\
76\text{–}87 & \text{Points} & \rightarrow & \text{Gut (2)} \\
88\text{–}100 & \text{Points} & \rightarrow & \text{Sehr gut (1)}
\end{array}
$$

0.9 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 Task 1.a: Multiplier

The first goal of this task is to implement a 4-bit multiplier as a Register-Transfer-Level (RTL) model with a separate state machine and datapath in the hardware description language SystemVerilog.

1.1 Understanding Scientific Practice

Before you get started, check the documents we provide on plagiarism. It is part of Task 1.a to read these documents and to acknowledge them.
1. Understand what is plagiarism. 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 0.9 and in the lecture slides.

1.2 Shift-and-Add Algorithm

Multiplication is a non-trivial operation and there exists no generic algorithm that provides all desirable properties. In this task, you need to implement a simple multiplier using the *shift-and-add algorithm* in a dedicated module. To explain the algorithm, consider the following example. Let A be binary 1011 and let B be binary 0110. Then 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 consider bits of A from least significant to most significant 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 gives our final result. Indeed, $6 \cdot 11 = 66$.

1.3 Specification

Use the program *Icarus Verilog* to develop an RTL-model of the multiplier algorithm using the hardware description language SystemVerilog. Develop a synchronous finite state machine (FSM) with an asynchronous reset to compute the multiplication of two 4-bit values using the shift-and-add algorithm. Implement this algorithm in the file `multiplier.sv` with an FSM and a separate datapath.

The top-level SystemVerilog interface is described in Figure 1.1. Initially, after the reset, the control signals `busy_o` and `valid_o` are 0. Only when `busy_o` is 0, the FSM reacts to the signal `start_i`. When `start_i` is set to 1, the computation starts, indicated by `busy_o` = 1. When the computation is finished, the control output `valid_o` is set to 1, the computed result is applied to the data output, and `busy_o` is set back to 0. The FSM remains in the *Done* state until a new computation is started. In addition to that, there is the obligatory clock signal `clk_i` and the reset signal `rst_i`. Figure 1.1 illustrates the top-level module in the `multiplier.sv` file.

```systemverilog
module multiplier(
  input logic        clk_i,
  input logic        rst_i,
  input logic        start_i,
  input logic [3:0]  a_i,
  input logic [3:0]  b_i,
  output logic       busy_o,
  output logic       valid_o,
  output logic [7:0] result_o
);
```

Figure 1.1: Top-level module of the Multiplier in SystemVerilog.

- `clk_i` is the clock signal for all registers.
- `rst_i` is the reset signal which is used to reset all registers.
- `start_i` is used to start the multiplication.
- `a_i` is the first non-negative input value.
- `b_i` is the second non-negative input value.
- `busy_o` indicates that the FSM is computing the multiplication.
- `valid_o` indicates that the computation is finished.
- `result_o` contains the computed multiplied value when the `valid_o` signal is high.

The FSM controls the datapath using dedicated control signals to perform the multiplication. The ASM diagram in Figure 1.2 summarizes the necessary arithmetic operations

and also shows the transitions between the different states. All arithmetic operations, including all counters, must be explicitly modelled within the datapath and are controlled via the FSM. Note, your hardware design must not contain *latches*. Use the command `make synth` to synthesize your HDL code to hardware using Yosys. The resulting area log output contains information on 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`).

1.4 Deliverables

All files must be submitted in folder `task-1a` of your repository. All files of the upstream repository must be included!

1. After reading content according to Section 1.1, 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. Edit the `README.md` file and describe which parts of your submission are (in-)complete to give your TA an overview.

3. Submit your SystemVerilog module for the multiplier in `multiplier.sv`.

   Add all files to the git repository. Make sure to <u>commit</u> your files and <u>push</u> them to your git repository on GitLab. Also, don't forget to create a <u>tag</u> and <u>push</u> it according to Section 0.6.

1.5 Hints

- Build a *synchronous* FSM with an asynchronous reset for the shift-and-add algorithm and integrate the required arithmetic into `multiplier.sv`. **Asynchronous designs, clock manipulation, or the use of non-synthesizable statements lead to a significant deduction of points.**
- **Separate** the datapath from the FSM. **Including the arithmetic operations within the FSM lead to a significant deduction of points.**
- Run `make` to build the multiplier module and its testbench.
- We provide a testbench, which instantiates the multiplier module. The testbench reads the input file `testcases_multiplier.txt`, applies the read values and control signals to the accelerator, and writes the results to the file `output_multiplier.txt`. Run `make test` to use the provided test cases.
- Add your own test cases to `testcases/testcases_multiplier.txt` to test your implementation.
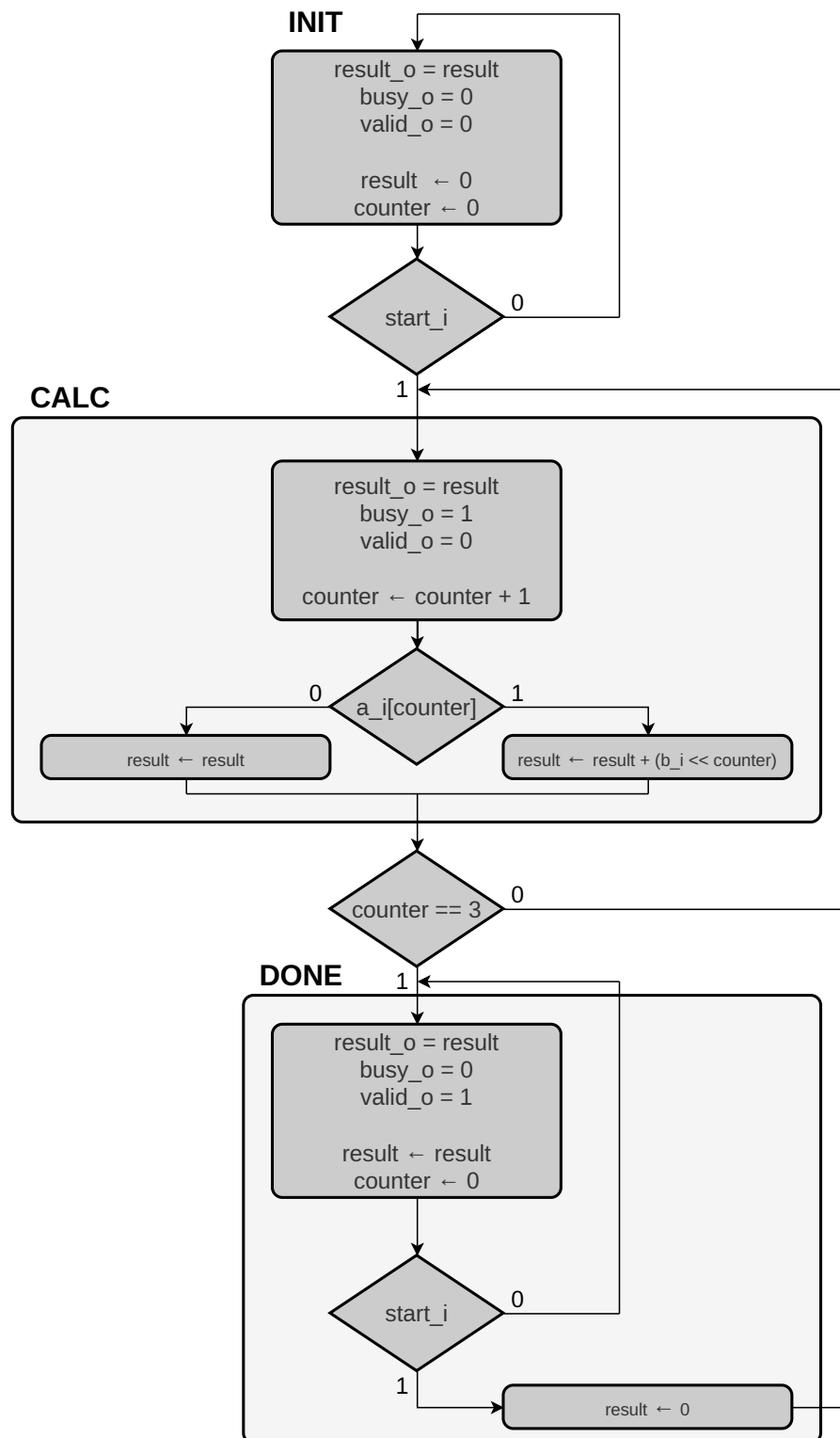
Figure 1.2: ASM diagram of the multiplier.

- For debugging purposes, it is helpful to visualize the timing behavior of the signals. For this, you can use the program *gtkwave*. The signals will be written to a `VCD` file. Run `make view` to view its waveforms.
- Run `make synth` to synthesize your HDL code to hardware using Yosys. **Hardware designs containing latches lead to a significant deduction of points.**
- 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.

2 Task 1.b: Binary GCD

The goal of this task is to understand the concept of a FSM and implement Stein's algorithm as RTL model. In this task, we use the Binary GCD algorithm, also known as *Stein's* algorithm, and implement it in hardware using the hardware description language SystemVerilog. The assignment repository already contains a test environment, including the testbench for the hardware design. Your task is to derive an ASM diagram of the state machine from the given pseudo code and implement the binary GCD in hardware.

2.1 Binary GCD Algorithm

The binary GCD algorithm, also known as Stein's algorithm, computes the greatest common divisor (GCD) of two non-negative integers. Compared to the Euclidean algorithm, Stein's algorithm uses comparisons, subtraction, and arithmetic shifts as a replacement for division operations. Algorithm 1 illustrates the pseudo code for the binary GCD algorithm. First, the algorithm checks whether one of the input values (or both) are zero and returns the according GCD value, i.e., `gcd(a, 0) = a`, `gcd(0, b) = b`, and `gcd(0, 0) = 0`. Second, the algorithm calculates 2^k, the greatest power of 2 value that divides a and b. Third, the algorithm divides a by 2 using shift operations until a becomes an odd number. Afterwards, the algorithm divides b by 2 using shift operations until b becomes an odd number. Depending on the values of a and b, the values potentially get swapped, and b is subtracted by a. This process of division, potential swapping, and subtraction is repeated until b is zero. Finally, we calculate the GCD value using a by restoring the power of 2 factor by shifting the computed result by the factor k.

2.2 Specification

Our goal is to implement the binary GCD algorithm in hardware using the hardware description language SystemVerilog. Therefore, you should derive an algorithmic state machine (ASM) diagram from the given pseudo code. First, you need to divide the pseudo code into different states. Second, you need to design the datapath based on the states of the ASM chart. Third, add the register transfer statements and outputs for each state. Finally, add the control logic for all state transitions and condition checks to the ASM diagram. Afterwards, you can start to implement the state machine in SystemVerilog using your ASM diagram.

Use the program *Icarus Verilog* to develop an RTL-model of the binary GCD algorithm using the hardware description language SystemVerilog. Develop a synchronous FSM

Algorithm 1 Binary GCD algorithm pseudo code

---

procedure BinaryGCD(a, b)
    if $a = 0$ **then**
        return b
    end if
    if $b = 0$ **then**
        return a
    end if
    $k := 0$
    while $((a \vee b) \wedge 1) = 0$ **do**
        $a := a \gg 1$
        $b := b \gg 1$
        $k := k + 1$
    end while
    while $(a \wedge 1) = 0$ **do**
        $a := a \gg 1$
    end while
    do
        while $(b \wedge 1) = 0$ **do**
            $b := b \gg 1$
        end while
        if $a > b$ **then**
            $tmp := a$
            $a := b$
            $b := tmp$
        end if
        $b := (b - a)$
    while $(b \neq 0)$
    return $a \ll k$
end procedure

---

with an asynchronous reset to compute the GCD of two values using the binary GCD algorithm. Your FSM has two 32-bit data inputs, `a_i` and `b_i`, and a 32-bit data output signal named `result_o`. Furthermore, the FSM has a 1-bit control input `start_i` as well as the two 1-bit control outputs `busy_o` and `valid_o`. In addition to that, there is the obligatory clock signal `clk_i` and the reset signal `rst_i`. Figure 2.1 illustrates the top-level module in the `gcd.sv` file.

```systemverilog
module gcd (
  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       valid_o,
  output logic [31:0] result_o
);
```

Figure 2.1: Top-level module in SystemVerilog.

- `clk_i` is the clock signal for all registers.
- `rst_i` is the reset signal which is used to reset all registers.
- `start_i` is used to start the GCD computation.
- `a_i` is the first non-negative input value.
- `b_i` is the second non-negative input value.
- `busy_o` indicates that the FSM is computing the GCD.
- `valid_o` indicates that the computation is finished.
- `result_o` contains the computed GCD value when the `valid_o` signal is high.

Initially, after the reset, the control signals `busy_o` and `valid_o` are 0. Only when `bus_oy` is 0, the FSM reacts to the signal `start_i`. When `start_i` is set to 1, the computation starts and leaves the initial state. This is indicated by the transition of the `busy_o` from 0 to 1. When the computation is finished, the control output `valid_o` is set to 1, the computed result is applied to the data output, and `busy_o` is set back to 0. Afterward, the FSM transitions to the initial state and waits until a new computation is started.

To test your circuit for correctness, you get a test vector and the corresponding result vector stored in `testcases_gcd.txt`. Use this data to verify the correctness of your **SystemVerilog** and implementation by adding additional test cases. The test vector is easily extendable using the format `<a b gcd(a, b)>`, e.g., `54 24 6`. The testbench is reading data linewise from this input file which must follow the aforementioned format. Each line consists of the two input values and the GCD result. The testbench applies the input data and the needed control signals to the instantiated module. Then, it waits until the computation finishes. Finally, the output value of the hardware design

is written to the output file `output_gcd.txt`. Suppose no value is read from the input file, the testbench and the simulation finishes.

Note, your hardware design must not contain *latches*. Use the command `make synth` to synthesize your HDL code to hardware using Yosys. The resulting area log output contains information on 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`).

2.3 Deliverables

All files must be submitted in folder `task-1b` 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 (in-)complete to give your TA an overview.

2. The ASM diagram of your FSM. The filename is `asm_diagram.pdf`

3. Submit your SystemVerilog module for the binary GCD in `gcd.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 0.6.

2.4 Hints

- Build a *synchronous* FSM with an asynchronous reset for the binary GCD algorithm and integrate the required arithmetic into `gcd.sv`. **Asynchronous designs, clock manipulation, or the use of non-synthesizable statements lead to a significant deduction of points.**
- Run `make` to build the GCD module and its testbench.
- We provide a testbench, which instantiates the GCD module. The testbench reads the input file `testcases_gcd.txt`, applies the read values and control signals to the accelerator, and writes the results to the file `output_gcd.txt`. Run `make test` to use the provided test cases.
- Add your own test cases to `testcases/testcases_gcd.txt` to test your implementation.
- For debugging purposes, it is helpful to visualize the timing behavior of the signals. For this, you can use the program *gtkwave*. The signals will be written to a VCD file. Run `make view` to view its waveforms.
- Run `make synth` to synthesize your HDL code to hardware using Yosys. **Hardware designs containing latches lead to a significant deduction of points.**
- 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.
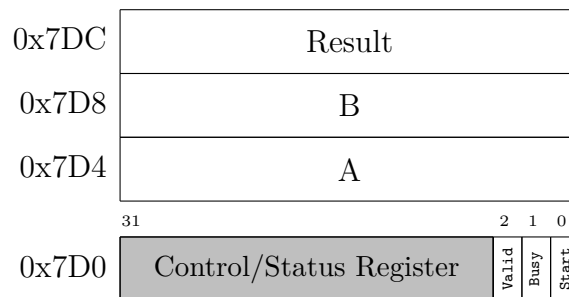
3 Task 2.a: GCD Co-Processor

In this assignment, you integrate the GCD accelerator from the previous task to the single-cycle MicroRISC-V CPU as a memory-mapped hardware accelerator. In the first part you integrate the peripheral as an accelerator to the CPU subsystem that can be polled. To be more efficient, the second task instructs you to enhance the CPU and the accelerator with interrupt support to avoid a costly polling operation.

3.1 Accelerator Design and Integration

The simple memory-mapped hardware accelerator contains four registers, a control and status register, the 32-bit input registers for A and B, and the result register. Build a wrapper around the GCD implementation from the previous task and build a memory-mapped interface for the described register set below. Integrate the accelerator into the CPU subsystem in `riscv_core.sv` and ensure the peripheral is only accessed for the memory map described below.

The control and status register itself contains three bits, described as follows:

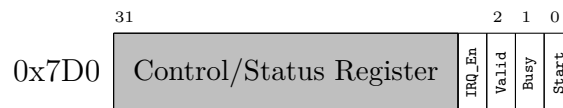| Bit	Name	Description
0	Start	Write 1 to start the computation. Starting the device while it's still busy does not have an effect. This bit always reads 0.
1	Busy	Reading this bit as 1 indicates the peripheral to be busy and cannot be be started again. When returning 0, the peripheral accepts starting a new operation.
2	Valid	Reading this bit as 1 indicates the operation has finished. Starting a new operation clears this bit.

The general operation of this accelerator is the following. First, the software writes the input values for A and B to the data registers. Second, the control register is written

with the `Start`-bit set to `1`. Third, the control register is polled until the `Valid` bit in the control register is set and the `Busy` bit is cleared. Finally, the software can read the data register to retrieve the GCD value.

To test the functionality of the peripheral, use the program `gcd_simple.asm`. This program is already provided inside the `testcases` folder.

3.2 Interrupts in RISC-V

Configuring the accelerator and polling the peripheral takes a long time, during which the CPU cannot perform other computations. To deal with this situation and to make the accelerator more efficient, you implement a simple interrupt system to the MicroRISC-V CPU and the peripheral. Therefore, you need to change the hardware and implement interrupt support. The interrupt is triggered via the signal `irq_o`, which is set to `1` for a single clock cycle. In the first step, extend the control register to support the interrupt configuration according to the specification below. Furthermore, assign the interrupt according to the configuration from the control register.

The control and status register itself contains three bits, described as follows:

Bit	Name	Description
0	`Start`	Write `1` to start the computation. Starting the device while it's still busy does not have an effect. This bit always reads `0`.
1	`Busy`	Reading this bit as `1` indicates the peripheral to be busy and cannot be be started again. When returning `0`, the peripheral accepts starting a new operation.
2	`Valid`	Reading this bit as `1` indicates finished its operation. Starting a new operation clears this bit.
3	`IRQ_En`	Setting this bit to `1` enables the interrupt when finishing an operation. Clearing it disables the generation of interrupts. Reading the bit indicates if the interrupt is enabled (`1`) or not (`0`).

When an interrupt occurs, store the current value of the program counter to a temporary register and jump to the interrupt service routine (ISR). In MicroRISC-V, the ISR is located at the fixed address `0x00`. Continue with the execution from the ISR. Note that registers are not saved through hardware, they must be handled in software, *i.e.*, needed registers are stacked.

To return from an interrupt, RISC-V defines the `MRET` instruction as defined below. This instruction updates the program counter with the previously saved program counter value, that was saved when the interrupt happened. Integrate this instruction into the MicroRISC-V CPU.

	31	20 19	15 14	12 11	7 6	0
MRET	001100000010	00000	000	00000	1110011	

To test the functionality of the interrupt implementation, use the program `gcd_irq.asm`. This program is already provided inside the `testcases` folder.

3.3 Deliverables

All files must be submitted in folder `task-2a` of your repository. Note, your hardware design must 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, *i.e.*, there is line similar to `$_DLATCH_N_8`.

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

2. Submit your modified CPU implementation including all files of the upstream repository.

3. The GCD peripheral from Task 3.1 must be implemented in `gcd_peripheral.sv`.

3.4 Hints

1. There are already assembly programs `gcd_simple.asm` and `gcd_irq.asm` in the testcases folder to test your implementation.

2. In case your GCD implementation from the previous task does not work, you can use the provided reference solution from the file `gcd_reference.sv`. This implementation, although synthesized, provides a functionally equivalent implementation to the task description.

3. 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.

4. Run `make view TARGET=<program-name>` to simulate your CPU and view the signal trace with GTKWave.

5. Run `make sim TARGET=<program-name>` to execute the program on the ISA simulator to observe the expected behavior.

4 Task 2.b: Binary Insertion Sort

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 Binary Insertion Sort algorithm and implement it in RISC-V assembly. The assignment repository already contains a reference implementation of the algorithm in C. First, you transform the existing C code into an assembly-like representation. Afterwards, you implement this algorithm in pure assembly and execute it on the RISC-V simulator.

4.1 Binary Insertion Sort Algorithm

The binary insertion sort algorithm is a sorting algorithm that, similar to the normal *insertion sort*, takes an unsorted array as input and returns the sorted array. However, the binary insertion sort applies a *binary search* instead of using a linear search to find the position where the element should be inserted. The sorting algorithm divides the array into a sorted and an unsorted sub-array. Starting with the first element of the array being in the sorted sub-array, the algorithm iterates from the second to the last element (the unsorted sub-array) and inserts the element into the sorted sub-array. The algorithm uses binary search to find the correct location for the insertion of the current element. The Wikipedia article provides a graphical visualization of the insertion sort algorithm and describes the variant using binary search. Furthermore, `binary_sort.c` contains the reference implementation of this algorithm.

4.2 Specification

Our goal is to implement the binary insertion sort algorithm using the RISC-V assembly language. To get started, you are given a complete implementation written in the C programming language (`binary_sort.c`). First, 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 an assembly instruction (`binary_sort_transformed.c`). Subsequently, the task is to convert the implementation to assembly (`binary_sort.asm`). The following description of `binary_sort.c` gives a short overview of the used functions.

- `main` allocates an integer variable `size` for the number of elements and an array containing the values to be sorted on the stack. Then it calls `input`, `insertion_sort`, and `output` in succession.
- `input` first reads the number of elements being processed from `stdin`. Then it reads this amount of values from `stdin` and stores it in the given array.

- **output** prints the sorted array to `stdout`.
- **insertion_sort** uses the `binary_search` function in order to process the received array by inserting all elements from the unsorted into the sorted sub-array.
- **binary_search** returns the correct location to insert the current element into the sorted sub-array.

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 several values corresponding to the number of elements. The output files consist of the sorted values of the processed array. Notice that 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/binary_sort.elf < test/input_01.testvec` or run `make test`, which provides a test suite.

binary_sort.c   This file provides you with a complete C implementation of the binary insertion sort algorithm. Use this file to understand the implementation.

binary_sort_transformed.c   In this file, the two functions `insertion_sort` and `binary_search` are not implemented. It is your task to implement these functions in such a way that each line in `binary_sort_transformed.c` corresponds to precisely one instruction in `binary_sort.asm`. This holds true for all lines except for the function entries and exits, which lead to corresponding prologues and epilogues in assembly. Note that the functions `binary_search` and `insertion_sort` must not have any parameters. Apply the RISC-V calling convention for passing parameters via the global registers to subroutines. For the implementation, mind the following rules:

- The functions `main`, `input` and `output` are already implemented. Use them and all other functions according to the RISC-V calling convention.
- Only use the registers declared at 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. The callee saved registers `s1–s11` are used to hold values that need to be restored after returning from a function call.
- The function `main` allocates the array and the local variable `size` 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.

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

binary_sort.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. Therefore, you must maintain the stack

for storing the return addresses, local variables, and spilled registers. Obey the following rules:

- The functions `main`, `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.
- Note, the callee saved registers `s1-s11` must be spilled on the stack before you can use them.
- Maintain a proper function prologue and epilogue.
- All array accesses must be resolved to dereferenced pointer accesses.

In this task, you use `riscvasm.py` for assembling the source code. This assembler has 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
- **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-2b` 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 (in-)complete to give your TA an overview.

2. Modify `binary_sort_transformed.c` to provide your transformed implementation of the functions `insertion_sort` and `binary_search`.

3. Modify `binary_sort.asm` to provide your assembly implementation of the functions `insertion_sort` and `binary_search`.

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 0.6.

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.
- Do not 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 3.a: Firewall

In this assignment, you will implement a network firewall in C.

5.1 Introduction

Whenever you browse the internet, play an online game, or watch a video, your computer communicates by sending *packets* of information to other computers. In this task, you will write a program that inspects these packets as they're being sent and received by your computer, and dynamically decides whether to allow them to be delivered.

This is what's commonly referred to as a "firewall". You will start by filtering based simply on the destination of the traffic, before working your way up to performing Deep Packet Inspection. This will even let you rewrite individual messages on the fly, as they're being transmitted.

5.2 Setup

We provide you with a Linux Kernel module that integrates with Netfilter[1] to forward packet data to userspace for processing. We also provide a userspace framework that retrieves the packets, and forwards a verdict (accept or drop) back to the kernel.

First, you'll want to make sure that the framework is functional on your machine.[2] To do this, follow these steps:

1. Open a shell in the `task-3a` directory

2. Run `make install` to build and install the kernel module

   - NB: this suspends all network traffic until you run the packet filter (in the next step)

3. Run `make run` to build and start the packet filter, which runs indefinitely

4. Run `ping 127.0.0.1` in a different shell

5. You should see output in the packet filter

6. CTRL-C to terminate the packet filter

---

[1] https://www.netfilter.org/

[2] If you are <u>not</u> using the provided CON VM, your kernel must be compiled with the `NETFILTER` flag.

7. Run `make uninstall` to remove the kernel module

   - This is needed to restore "normal" network traffic behavior

By default, the network "filter" stub simply prints out the raw packet before accepting. Your task is to actually implement the `processPacket` method, which can be found in `firewall.cpp`. It is passed the raw IPv4 frame, and returns either `ACCEPT` or `DROP`.

5.2.1 IPv4 subnet notation

IPv4 addresses are 32 bits in length, and are denoted as a tuple of four eight-bit unsigned integers, written as decimal numbers separated by dots (`.`).

Groups of IPv4 are commonly denoted in subnet notation. Subnet notation consists of an IP address, followed by a forward slash (`/`), followed by the number of leading bits that must match the specified IP address.

For example, the subnet `127.0.0.0/8` consists of all IP addresses whose first 8 bits equal the first 8 bits of `127.0.0.0` – that is, any IP address whose first byte is `0111 1111` (`127`). Similarly, the subnet `172.16.0.0/12` consists of all IP addresses whose first 12 bits equal the first 12 bits of `172.16.0.0` – that is, any IP address whose first byte is `172`, and whose second byte is of form `0001 xxxx`.

5.3 Your task

Your task is to implement the following filtering rules. Tasks are listed in order of difficulty, so it is recommended that you start at the top to get a feeling for things, before progressing down the list.

1. Any packet that is addressed to the exact IP address `8.8.4.4` must be dropped.

2. Any packet that is both *from* and *to* the loopback subnet (`127.0.0.0/8`) must be accepted.

3. Any fragmented IP packet must be dropped.

4. Any packet *not* using one of the following protocols must be dropped: ICMP, TCP, UDP

5. Every third outgoing ICMP Echo Request to a private address[3] must be dropped.

   - ICMP echo requests have type `0x08`.
   - Which particular packets you drop is irrelevant, as long as any sequence of $3n$ matching packets results in n packets being dropped.

6. Every fourth incoming ICMP Echo Request from a private address must be dropped.

---

[3]any address in any of `10.0.0.0/8`, `172.16.0.0/12`, or `192.168.0.0/16`

7. Any ICMP packet, except those dropped under previous rules, must be <u>accepted</u>.

8. Any outgoing TCP packet to port 443 should be <u>dropped</u>.
   - Try turning on HTTPS-only mode in the security settings of your browser. What happens if you now try to connect to a server?

9. Every second outgoing TCP packet to port 80 should be <u>dropped</u>.
   - Try browsing to a HTTP site. Notice how all data is still received? That's TCP at work!

Tie-breakers

Unless otherwise specified, earlier rules take precedence over later rules. For example, if a packet meets the criteria for both rule 2 and rule 4, rule 2 should be followed, and the packet should be accepted.

5.3.1 Bonus task: HTTPS SNI sniffing

If you choose to implement this bonus task, you will disregard rules 8 and 9 in the previous section. Instead, you will implement two additional features for filtering connections.

First, you will implement *modification* of packets, rather than just dropping them. To do this, you will first need to create your modified packet in memory. Then, pass it to one of the `setOverridePacketData` methods on an `ACCEPT` FilterResponse. Note that you will also need to update the TCP checksum appropriately, or the packet will not be accepted by the network stack.

You will use this to modify all *incoming* TCP packets originating from port 80. Modify the packet, replacing all occurrences of the case-sensitive ASCII string `Orange` with `Banana`.[4] This serves primarily as a test that your substitution mechanism is working. Suitable content to test your substitution on can be found at http://con-target.student.iaik.tugraz.at/fruits.html.[5]

Second, you will parse any *outgoing* packet to TCP port 443 as a potential SSL handshake ClientHello message. The record type for an SSL handshake is `0x16`, and the protocol type for ClientHello is `0x01`. If the packet consists of a ClientHello message, you will parse that message, looking for a Server Name Indication (SNI) extension block. This extension block carries the domain name that the client is connecting to.

We will only permit connections to exactly the domain `de.wikipedia.org`, or to the domain `tugraz.at` and any of its subdomains. For any other SNI domain, or if no

---

[4]You do not need to identify the HTTP response body. Replace any occurrences of the exact byte sequence `4F 72 61 6E 67 65` with `42 61 6E 61 6E 61`. The two strings are, conveniently enough, of the same length.

[5]If you wish to test on other websites, you may need to disable GZIP compression in your browser for a naive approach to work. To do this in Firefox, set `network.http.accept-encoding` to an empty string in `about:config`. You do not need to do this when using http://con-target.student.iaik.tugraz.at/fruits.html.

SNI extension is present, cause the connection to fail by overwriting all specified cipher suites with `0xeaea`. This will cause the server to abort the handshake. Note that you again will need to update the TCP checksum to match. Leave the rest of the message unmodified.

5.4 Deliverables

All files must be submitted in folder `task-3a` 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-3a`.

1. Implement the rules in `firewall.cpp`

2. Add any other required `.cpp` or `.h` files to the `task-3a` base folder *(no subfolders!)*

3. Edit the `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 <u>commit</u> your files and <u>push</u> them to your `git` repository on `GitLab`. Also, do not forget to create a <u>tag</u> and <u>push</u> it according to Section 0.6.

Your submission should run without crashing or leaking memory. Use the Valgrind-enabled target (`make valgrind`) in the provided `Makefile` and test your program thoroughly!

5.4.1 Testing

A copy of our testing harness is provided to you in the `task-3a-tester` folder of your repository. To start it, simply use `make run`. Your source files from the main `task-3a` folder will be automatically copied over.

Your deliverable should, at minimum, *run* in this testing harness. *(Ideally, it should also pass all test cases.)*

The test cases provided are only a subset of the test cases we will use to evaluate your deliverable, and should not be interpreted as a complete list of requirements. They are only provided for your convenience. Feel free to add additional test cases.

Any modifications you make to this folder in your submission will be ignored. If you have any issues with the testing harness, please raise them on Discord.

6 Task 3.b: Internet Radio

In this task, you will implement a basic web server that implements the Real-Time Streaming Protocol (RTSP). Your server will negotiate media streaming sessions with a client application. Once you are done, you will be able to play this stream using common media players, such as ffplay.

Our RTSP server listens for connections on TCP port **8554**. Once a connection is established, the client and server communicate using a text-based protocol. For this task, we will limit the protocol to only the functionality required to get basic playback in a media player.

Once negotiated via the RTSP connection, the server will send audio data to the client using the UDP-based Real-time Transport Protocol (RTP). You will not need to handle audio file formats in this task. Instead, reading and transcoding audio will be outsourced to the `libavcodec` and `libavformat` libraries, which are part of the FFMPEG suite.[1] Our framework provides wrapper functions that handle the interaction with `libavcodec` and `libavformat` so that you can concentrate on implementing the actual protocol.

6.1 The Real-Time Streaming Protocol

The real-time streaming protocol (RTSP) is a text-based protocol used to negotiate media streaming sessions between a server and one or multiple clients. Note that RTSP does, in general, not deliver the streamed media itself. Instead, a different protocol like RTP performs the actual streaming of media data. When using RTSP, all client sessions are uniquely defined by a *session identifier*. A client can close and re-open its connection to the streaming server multiple times without having to re-negotiate session information. RTSP is a stateful protocol, and each client session has a well-defined state at any point in time. It is advisable that even a simple RSTP server keeps track of the current state of each session. In the case of a single-client application, you only need to keep track of one session at a time. Each session can have the following states.

- **Initialization:** In the **initialization** state, no resources are currently allocated for the stream. This is the initial state of each connection. Upon receiving a setup request, the connection will change to the next state. From the initialization state, no other state transitions are possible.
- **Ready:** After allocating the resources needed for a new session the connection will be in the **ready** state. In this state, subsequent setup requests will overwrite the previous setup configuration, but the connection will remain in the ready state. In

---

[1]https://ffmpeg.org/documentation.html

our assignment, we assume that each client sends exactly one setup request. Hence you will not have to handle multiple subsequent setup requests. The connection will stay in the ready state until a play request or a teardown request is received. A play request will advance the connection state while a teardown request will reset the connection back to the initialization state.

- **Playing:** After receiving a play request and sending the appropriate response, the connection switches to the **playing** state. In this state, media data is actively transmitted to the client via the defined channel (the RTP stream in our case). The connection will switch back to the initialization state once all data has been transmitted. You will not need to handle requests while the connection is in the playing state. Once the requested file stream has finished, the connection shall free the resources and transition to the initialization state.

6.1.1 Request and Response Format

Each request and each response follows a strict format similar to that of HTTP messages. All lines are terminated by \r\n. Every request starts with a *request line* with the format *method*␣*path*␣RTSP/1.0[2]. The methods that your implementation must support are listed further below,. Requesting an unimplemented method should result in an appropriate error response (see Table 6.1). The *path* is the absolute URL of the requested resource that the server should stream to the client. In this implementation you will have to support paths of the form rtsp://<server ip>:<server port>/<resource name>. Requests that target a resource not present on the server will be met with an error message.

After the request line, the client will send any number of header lines of form *key*: *value*. The key is case-insensitive. An arbitrary amount of whitespace[3] may occur between the colon and the value, and must be ignored. The header section is terminated by an empty line.

When responding to client requests, the first line of the response is the *status line*. The status line has the form RTSP/1.0␣*code*␣*description* and is terminated by \r\n. Similar to the request line in the request format, the status line is followed by a number of header lines of the form *key*: *value*. In your implementation, some of the responses will carry an additional *entity*. In such cases, the header fields of the response must include the Content-Type and the Content-Length field. An entity is the body or the content of a message.

In RTSP, each message will carry a CSeq header field. This field contains a sequence number that is increased on each *distinct* request from the client. When handling client requests, the CSeq field of the server response will be equal to the value sent by the client.

While RTSP supports a multitude of possible status codes, your implementation will implement the subset of status codes listed in Table 6.1.

---

[2]␣ is a single whitespace, code point 0x20, *i.e.*, ' '

[3]we define *whitespace* to be exclusively code point 0x20, *i.e.*, ' '

Status Code	Description	Usage
200	OK	Used as a response to every successful request.
404	Not Found	The server cannot find the requested resource.
454	Session Not Found	The `session` header does not match the previously negotiated session identifier.
455	Method Not Valid in This State	The method of the request does not match the current server state.
501	Not Implemented	The method of the request is not implemented.

Table 6.1: Status codes supported by your application.

In your implementation you will not need to account for requests that miss required headers or are malformed.

6.1.2 The OPTIONS request

The client issues an OPTIONS request to receive a list of methods that are supported by the server. In our basic implementation, the server will support the methods OPTIONS, DESCRIBE, SETUP, PLAY and TEARDOWN. An options request may be received at any time and does not alter the server or the client state. When responding to an OPTIONS request, the response will contain the `Public` header field. In your implementation, the value of this field will be a comma-separated list of all supported methods. During an OPTIONS request, you may ignore the path of the request.

6.1.3 The DESCRIBE request

With the DESCRIBE method, the client requests the description of the media object that will be streamed from the server. This media description is given in the SDP format and must contain all initialization information for the stream. Note that the description can contain the target port that a stream will be sent to. This target port is only relevant when using the multicast mode of operation and, thus, is irrelevant in your implementation. Similar to the OPTIONS request, the DESCRIBE method does not alter the server state.

When responding to a DESCRIBE request, the server will send the SDP information as an entity in the response message. The SDP format contains the session description, the time description, and the media description of a stream. For an in-detail description of the SDP format, please consult RFC 2327.

As the SDP information is contained as an entity in the response, each response to a DESCRIBE must contain the `Content-Length` and the `Content-Type` header. The

transmitted content length is the size of the SDP information in bytes. In the response, a single line containing only \r\n separates the header and the entity.

Hint: You do not have to implement the generation of the SDP data by hand as libavformat supports the automatic generation of SDP files. Detailed knowledge on the SDP format is not needed to solve this task. If you are interested in how the SDP data is generated, please refer to RFC 2327 or the implementation of getSDPInfo in the framework.

6.1.4 The SETUP request

With the SETUP request, the client instructs the server to allocate resources for a media stream. The server will create a new session bound to a unique session identifier. The client will use the session identifier in all further requests. With a successful SETUP request, the server transitions from the initialization state to the ready state. Your implementation will only handle the unicast mode of operation. In this mode, the client will specify the ports on which the media data will be received. Make sure that your implementation actually uses the specified ports.

Each SETUP request contains a Transport header field. This field holds the acceptable parameters for the actual data transmission. The field content has the format RTP/AVP/UDP;unicast;client_port=<port pair>. The port pair defines the ports on which the client will receive media data and control information. The first port of the pair defines the media port, and the second is the control port. When receiving a port pair, the ports are divided by a single - character. Hence, the port pair has the format <media port>-<control port>. **When streaming via RTP in unicast mode you must use the client-specified media data port.**

A response to a SETUP request must contain a copy of the original transport definition. Additionally, the response will contain the session id that uniquely identifies the current session. You can use a monotonically increasing counter or a random number generator to generate unique session identifiers.

6.1.5 The PLAY request

A server that is in the ready state will switch to the playing state upon receiving a PLAY request. For that, the server will send a reply to the client and then start the actual media streaming via the transport method defined during the SETUP request. A PLAY request may contain a Range header field containing timing information in the *npt* format. Your implementation may ignore the field as we always stream the complete file, starting from the first frame until the end. Note that the Range header is optional.

The actual data transport will use RTP (cf. Section 6.2) and transmit the media via UDP. You can use the readPacketFromContext and sendAndFreePacket framework functions for that.

Your implementation should implement a simple rate-limiting when sending the packets. Use the duration of each packet to decide how long your implementation will wait until the next packet is sent. Make sure that you use the correct time units when doing

so. Take a look at the libav documentation to see how the duration can be extracted from a packet.

6.1.6 The TEARDOWN request

The TEARDOWN request causes the server to immediately stop the stream and deallocate all resources associated with the session specified in the request headers. A TEARDOWN invalidates the session identifier. Note, however, that a teardown is not equivalent to a disconnect from the client. After a TEARDOWN request, the client may issue further SETUP, DESCRIBE, or OPTIONS requests.

6.2 The Real-time Transport Protocol

The Real-time Transport Protocol (RTP) handles the transfer of stream data from the server to the client. While the RTSP logic has to be implemented by you, the RTP logic is handled by the `libavformat` library. At its core, RTP transports *payloads*, in our case the audio data, in *packets* using UDP as its underlying protocol. Our framework provides you the `readPacketFromContext` and `sendAndFreePacket` methods to read RTP packets from the input stream and send them to the client. For a detailed description of RTP consult RFC 3550.

6.3 Acceptable Limitations

We understand that base task 3b is already quite complex. Therefore, the following limitations in your submission are explicitly acceptable.
- You do not need to handle multiple clients.
  Only the first client to connect on port 8554 needs to be handled.
- You do not need to handle multiple subsequent SETUP requests.
  Each client connection will send exactly one SETUP request.
- You do not need to handle further RTSP messages after PLAY.
  This lets you switch over to RTP handling entirely at this point.
- You do not need to implement complex session management.
  We assume that each session is terminated after leaving the playing state.
  We will not test your implementation against interleaved requests with different session identifiers.

6.4 Bonus task: Pausing streams and multiple clients

The limitations above are not something you would allow in actual server software. Therefore, in this bonus task, you will adjust your solution to remedy them.

6.4.1 Pausing streams

You will also need to handle the following RTSP command, which may be sent by the client during playback. While it is possible to use threads to handle both tasks (sending RTP data and monitoring for RTSP messages) at the same time, it is likely *not* the easiest approach. Instead, we recommend that you set your RTSP socket to non-blocking mode.

This allows you to use `read` to check whether data is available, without execution being suspended if there isn't. Use this to check for messages periodically before going back to processing RTP data. Alternatively, you can use `ioctl` to query if there is pending data on the socket.

The PAUSE request

When receiving a PAUSE request, the server will stop the stream at the current point in time, i.e., the current packet. The server must "remember" the point at which the stream was paused. Receiving a subsequent PLAY request will cause the stream to continue from the point at which the playback was paused. Receiving a TEARDOWN after a PAUSE will free all resources and invalidate the session.

6.4.2 Multiple clients

You will need to handle multiple clients' RTSP/RTP connections, possibly at the same time, while also checking back to `accept` new incoming connections periodically. It is possible to do this in a single thread: accept connections, then loop through all currently-open connections, poll their RTSP sockets for messages, and stream a frame of RTP data. Then, repeat.

Of course, it may well be easier to use a dedicated thread for each connection you accept. The details are left to you. Make sure that you keep track of all alloated `AVContext` to avoid memory leaks!

6.5 Deliverables

All files must be submitted in folder `task-3b` 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, they will simply be ignored on the test system.

1. Edit the `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 0.6.

Your submission should run without crashing or leaking memory. Use the Valgrind-enabled target (`make valgrind`) in the provided `Makefile` and test your program thoroughly!

6.5.1 Testing

A copy of the framework that we use for testing can be found in the `task-3b-tester` folder of your repository. As in the previous task, simply use `make run` to start the test suite. Your source files from the main `task-3b` folder will be automatically copied over.

Again, we provide you with a subset of all test cases. The test cases that are provided to you only cover the most basic functionality. Thus, you should write your own test cases to further test your implementation.

Any modifications you make to this folder in your submission will be ignored. If you have any issues with the framework, please contact us on Discord.

6.5.2 Hints

- As an alternative to the test suite, you can test your implementation using `curl`. Adapt the command `curl -i -X OPTIONS 127.0.0.1:8554/sample.mp3 -H "CSeq: 1234" --verbose --http0.9` to your needs.

- You will need little to no dynamic memory in your implementation. Make sure to check your implementation for memory leaks using the `make valgrind` build target.

- For the rate-limiting you can use the `duration` field found in each packet. However, the duration must be scaled to seconds using the time-base of the packet.

Errata

This section lists releases and changes of this file.

2022-**10**-**06** Initial release.

2022-**10**-**13** Fix abort condition in steins algorithm.

2022-**10**-**20** Clarify Task-2a. Remove assembly program from task to ease it.

2022-**11**-**07** Clarify Task-3a.