Computer Organization and Networks Practicals 2021/22

December 6, 2021

Contents

0 Introduction

This document describes the tasks for the course "Computer Organization and Networks Practicals" for the winter term 2021/22. In this course, we are going to study computer architectures and networking stacks. We will discuss how CPUs are designed and how we can speed them up. Using CPUs as building block, we can build applications on top. One crucial component is allowing these CPUs to interact. In the network part, we look at the TCP/IP stack which defines our networks today.

Before presenting the assignments, we cover all organizational parts:

- Registration
- Assignment sheet
- Communication channels
- Tutorial videos
- Toolchain
- Question hours
- Submissions
- Task interviews
- Grading
- Plagiarism

0.1 Registration

The registration in TUGRAZOnline for this course ends on 2021-10-07. Please register for one of the ten groups. If you don't have a TUGRAZOnline account yet, please contact us before 2021-10-07 via con@iaik.tugraz.at.

If you participate in any submission process, you are going to get a grade at the end of the semester.

0.2 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) is provided with your repository. Fetch updates to receive the latest version:

```
git pull
```

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

0.3 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 civil name, it helps your TA to recognize you. To join at Discord, you can use the following invitation link:

`https://discord.gg/mxuUnjP`

- `#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. Teaching Assistants (TAs) might attend but do not have to answer questions here.
- `#con-ta` is a prefix used for audio-only channels per TA. During their respective question hour, you can ask questions here and your TA will answer them.

0.4 Tutorial videos

Date	Content	Video
2021-10-06	Getting started	on seafile
2021-10-06	Task 1.a	on seafile
2021-10-06	SystemVerilog	on seafile
2021-10-06	Task 1.b	on seafile
2021-11-05	Task 2.a	
2021-11-05	Task 2.b	
2021-12-03	Task 3.a	
2021-12-03	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. The link will be shared on `#con` and in the newsgroup. The main goal is to introduce students to the next task and show them the required toolchain.

0.5 Toolchain

The toolchain is introduced in tutorial videos (Section 0.4), but we still want to document it here once more. The entire software stack, occuring in these practicals, is given by:

- git for version control (and a GitLab server for submissions)

- Digital (→ Digital homepage)
- 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
- The C and C++ programming languages

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 students with installing software, we provide a virtual machine for VirtualBox. The virtual machine has the entire software stack preinstalled and is based on Ubuntu 20.10:

`https://seafile.iaik.tugraz.at/f/d91aa405a7584e7c847f/`

0.6 Question hours

	Monday	**Tuesday**	**Wednesday**	**Thursday**	**Friday**
08:00	Martin		Stefan	Katrin	
09:00	Matthias	Marcel		Fabian	
14:00	Niklas				
16:00		Ferdinand	Moritz		
17:00					Constantin

Table 0.2: TA weekly question hour times.

Question hours are specific for your TA. They take place every week and you can look up your TA's day and time in Table 0.2. In the `#con-ta` channel of your TA, you can ask any questions about the tasks during the one hour question time.

0.7 Submissions

At the beginning of the semester, you will receive account credentials for some GitLab instance. Using git, you can submit your deliverables in your personal git repository. To submit, 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. 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 latest possible submission deadlines for the respective tasks are given in Table 0.3. These timestamps are hard deadlines. If you submit your solution after a deadline, you will lose 8 points (from your achieved points on the respective task) per started 24 hours.

Task	**Deadline**	**Max. points**
Task 1.a (Divider)	Fr, 2021-10-29 23:59	max. 15 points
Task 1.b (Divider and CPU integration)	Fr, 2021-11-05 23:59	max. 20 points
Task 2.a (Pipelining CPU)	Fr, 2021-11-26 23:59	max. 20 points
Task 2.b (Quicksort in RISC-V)	Fr, 2021-12-03 23:59	max. 15 points
Task 3.a (Switching)	Fr, 2022-01-14 23:59	max. 10 points
Task 3.b (A tiny HTTP server)	Fr, 2022-01-21 23:59	max. 20 points

Table 0.3: Task submission deadlines and maximum achievable points.

0.8 Task Interviews

There are going to be two task interviews (for three tasks each). The dates for the interviews will be organized by your TA and he/she will inform you ahead of time.

The interviews cover general questions of each topic and also discuss your particular solution you submitted. For your task interview, please 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 interviews is to verify you did the implementation yourself, understood the topic and collect feedback on both sides.

0.9 Grading

The maximum points per task are listed in Table 0.3. Depending on your achieved points, you will get the associated grade according to Table 0.4.

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)

Table 0.4: Points to grade mapping.

0.10 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!
- Commit regularly to show activity!

1 Task 1.a: Divider

In the first task, you will build a division machine, which is capable of dividing some 4-bit dividend by some 4-bit divisor. This should give you a decent introduction to digital hardware design. Hardware design is usually done in hardware description languages like VHDL or Verilog.

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 confirm them.

- Understand what is plagiarism. You can find more information on this topic on plagiarism.org.
- Study the document "Guidelines on Safeguarding Good Scientific Practice".
- Understand the consequences described in Section 0.10 and in the lecture slides.

1.2 Interface of your divider module

We recommend to implement this exercise with Digital, but we only test the Verilog export. As such, you can also solve the exercise in Verilog or SystemVerilog. The interface of your divider module must support the following signals:

signal	direction	bit width
clk_i	input	1
reset_i	input	1
dividend_i	input	4
divisor_i	input	4
start_i	input	1
busy_o	output	1
finish_o	output	1
quotient_o	output	5

clock_i and reset_i signals are input signals for every synchronous machine. dividend_i, divisor_i, and quotient_o relate to the division operation. start_i, busy_o, and finish_o are status signals indicating the progress of the computation.

In Figure 1.1, you can see the top level interface in Digital. Equivalently, the same interface in Verilog is shown in Figure 1.2. Since you need to submit a Verilog module, you have to implement the interface of Figure 1.2.

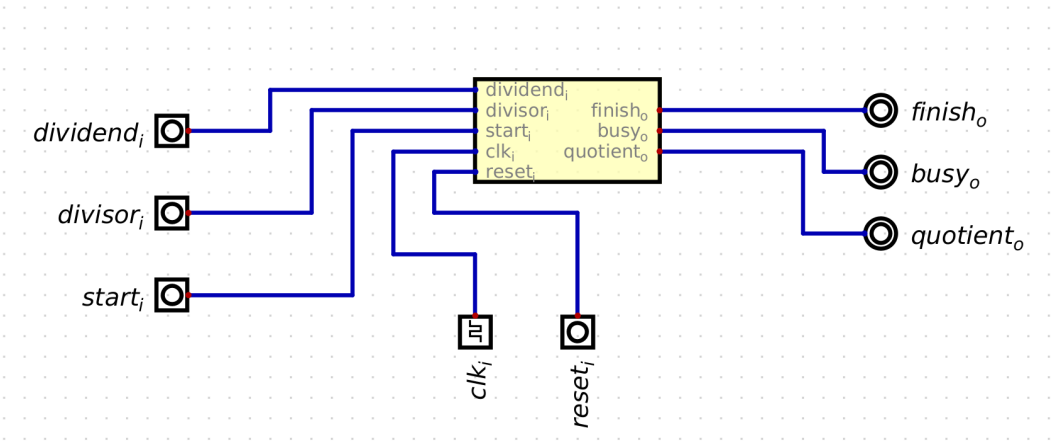

Figure 1.1: High-level view of the Divider in Digital.

```
module divider (
  input clk_i,
  input reset_i,
  input [3:0] dividend_i,
  input [3:0] divisor_i,
  input start_i,
  output busy_o,
  output finish_o,
  output [4:0] quotient_o
);
```

Figure 1.2: Top-level module in Verilog.

1.3 Algorithm

Algorithmically, we are going to use the division by subtraction algorithm. In this approach, we subtract the divisor from the dividend per clock cycle until the dividend is smaller than the divisor. At the same time, each iteration increments a temporary variable. This way, we count the number of subtractions which gives us the quotient which satisfies $0 \leq \text{dividend} - \text{quotient} \cdot \text{divisor} < \text{divisor}$.

As a result, you have to use the following registers:

state The current state of the finite-state machine. You have to use three states. We call them INIT, BUSY, and FINISH. You need to give them binary numbers on your own.

quotient This is the above-mentioned temporary variable. It increments with each subtraction and thus gives the final quotient.

subtrahend The subtrahend is initialized with the dividend and subtractions are applied together with divisor_i

In Figure 1.3, you can find the ASM diagram of the algorithm to implement. Be sure to follow the names of the diagram for internal signals.

1.4 Specification

Your circuit must fulfill the following specification:

- We have input and output signals. In the submission, the order matters (the order is given in the Verilog interface Figure 1.2). You can reorder signals in Digital with "Edit / Order Inputs" and "Edit / Order Outputs".
- 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 dividend_i and B be the value of signal divisor_i
  2. after a finite amount of clock cycles, let finish_o become high.
  3. if finish_o is high,
     - if B was 0, quotient_o must have value 31
     - otherwise quotient_o must have value $\lfloor \frac{A}{B} \rfloor$.
- Use the "D-Flip-flop, asynchronous" as a register. Connect the asynchronous reset to reset_i.
- You can assume that between start_i becoming high and finish_o becoming high, the value divisor_i does not change.
- Your next state logic must be implemented as a gate-level netlist (i.e. do not use multiplexers in Digital and equivalently do not use `case` or `if` keywords in SystemVerilog)
- You need to submit one Verilog file with your implementation. The top-level module must be called "divider". In Digital, you can export the implementation with "New / Export / Export to Verilog". The top-level module name is defined by the export filename without file extension.

1.5 Testing

Once, you are finished with your implementation, store it as file `ips/divider.v`. Then use the `make run_divider` command to execute the testbench. If the provided three testcases pass, the message "All tests completed successfully!" will be printed. Otherwise the testbench will terminate with the first failing testcase.

Be aware that passing three testcases does not mean your implementation is correct and will give you all points.

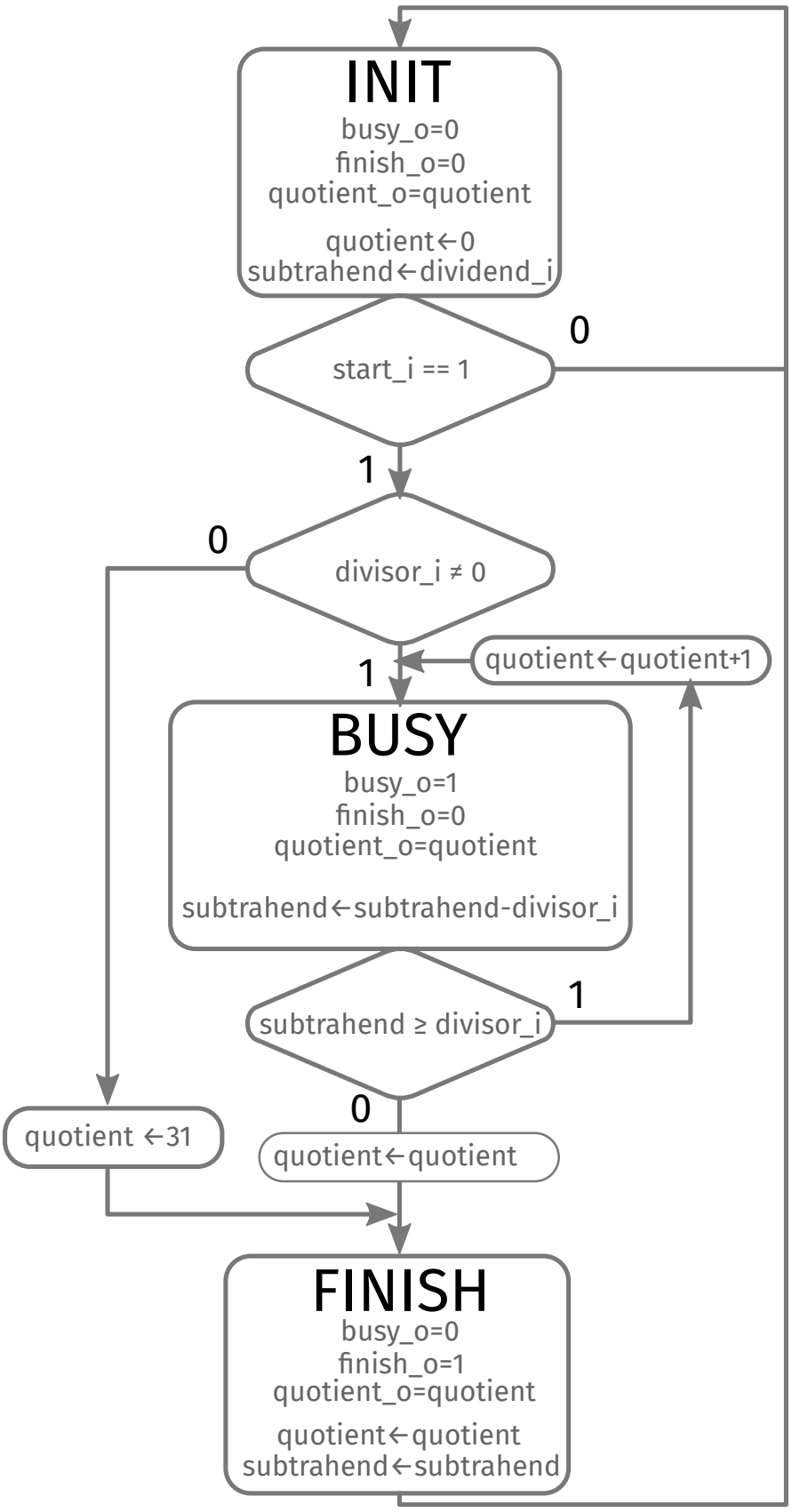

Figure 1.3: ASM diagram of the divider.

1.6 Deliverables

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

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 understood the consequences,
   - and that you won't submit plagiarized work.

   Add this file to your git repository.
2. The main circuit needs to follow the specification mentioned in Section 1.4. Submit your solution in file `ips/divider.v`. You might also want to add your Digital files. If there are issues, your TA can give you better feedback if you submit your Digital source files.
3. Edit the `README.md` file and describe which parts of your submission are complete to give your TA an overview.

IMPORTANT Make sure to commit your files and push them to GitLab.

IMPORTANT Don't forget to create a tag and push the tags according to Section 0.7.

2 Task 1.b: Divider and CPU Integration

The first goal of this task is to implement an 8-bit divider as Register-Transfer-Level (RTL) model in the hardware description language SystemVerilog. Use the same algorithm from Task 1.a, but extend its datawidth to 8-bit. In a second step, you integrate this divider to the MicroRISC-V CPU by adding a division instruction according to the RISC-V instruction set.

2.1 Divider

Implement the divider from Section 1 as RTL model. Increase the bitwidth of the data inputs to 8-bits and the bitwith of the quotient output to 9-bits, to support 8-bit division operations. Note, when a division is started with the dividend being zero, the output should be -1 *i.e.*, all bits set to one.

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 2.1 shows the microarchitecture of the implemented CPU. Your task is to extend the CPU with the `DIVU` instruction of RISC-V's M extension.

Extend the decoder of MicroRISC-V to support the `DIVU` instruction using the instruction information given in Table 2.1.

`DIVU rd, rs1, rs2.` Perform a 8-bit/8-bit division between rs1 and rs2 and store the 8-bit result in register rd. If the dividend is zero, the result in register rd should be -1, *i.e.*, all bits set. Do not forget to sign-extend the division result to 32-bits!

Note, in RISC-V, the `DIVU` instruction would perform a 32-bit unsigned division. In the practicals, the data size is reduced to 8-bits!

Integrate the divider to the given MicroRISC-V CPU by instantiating the divider within the CPU. Extend the decoding and executing steps of the CPU for the divider. Note, the division operation is a multi-cycle operation. Thus, it requires to stall the CPU until the division algorithm finishes and its result is available.

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

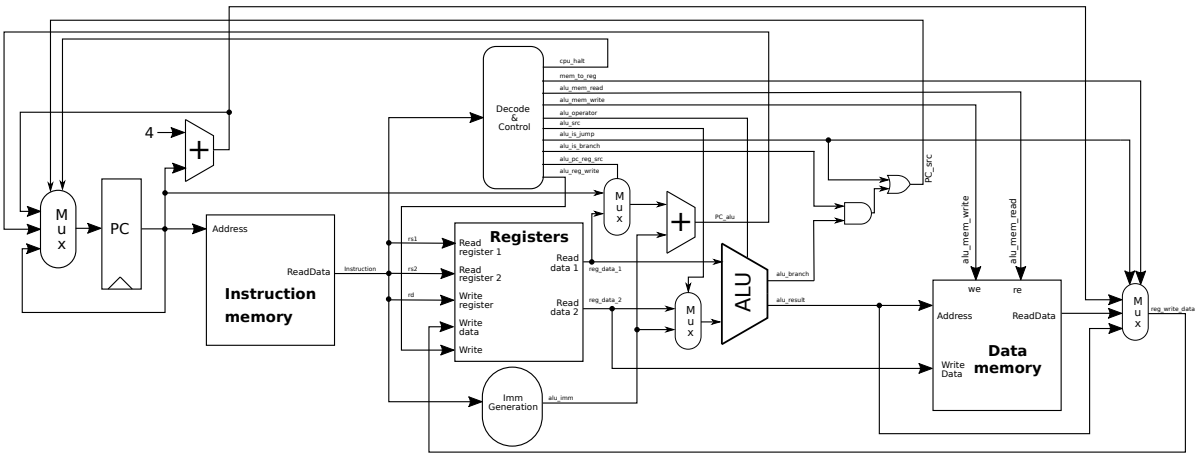

Figure 2.1: MicroRISC-V architecture.

Table 2.1: Encoding of the division instruction to be implemented.

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0	
0000001	rs2	rs1	101	rd	0110011	DIVU

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`).

2.3 Deliverables

All files must be submitted in folder `task-1b` 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 Verilog module for the divider in `ips/divider.sv`.
3. Submit your modified MicroRISC-V implementation including all files of the upstream repository. This implementation is supposed to feature the `DIVU` instruction.

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 0.7.

2.4 Hints

- Run `make divider` to build the divider and its isolated testbench, run `make run_divider` to run it, and `make view_divider` 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 the `DIVU` instructions. Use the `-e` flag as follows: `riscvasm.py -e div_extension.py testcase.asm > testcase.hex`. The Makefile does it automatically for you.
- 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: Pipeline a RISC-V CPU

In this task, you improve the performance of the single-cycle MicroRISC-V CPU, by pipelining the processor. In particular, you will take the MicroRISC-V from Task-1b, and transform that to a 4-stage pipeline. Although the single-cycle CPU works correctly, it is too inefficient for practical usage in modern designs. The longest path in the design determines the clock frequency, thus limiting the performance of the overall design. To cope with that problem, and to make the design more efficient, pipelining is applied. With pipelining, multiple instructions are overlapped in the execution being executed in different stages.

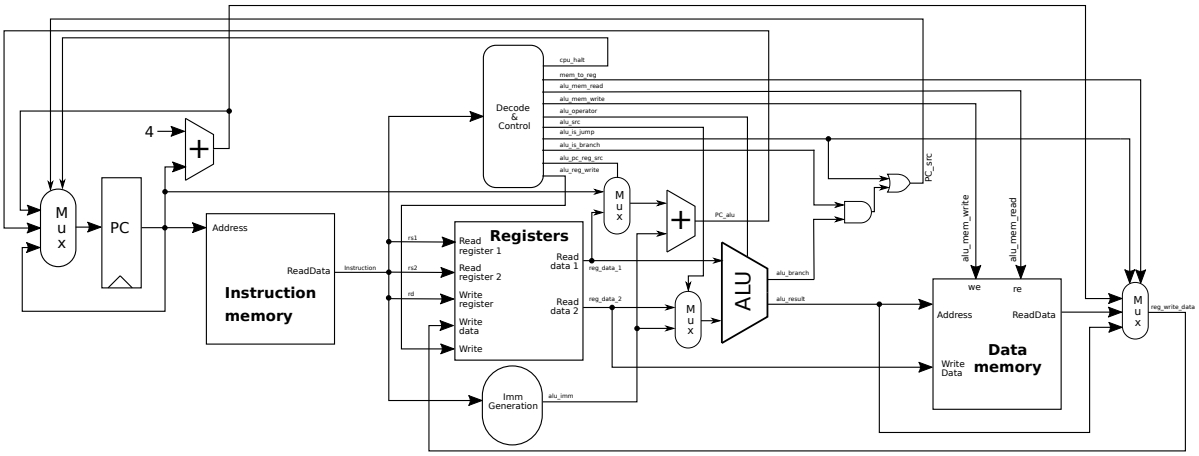

Figure 3.1: Starting Point: Single-cycle MicroRISC-V CPU.

The starting point for this task is the single cycle processor of MicroRISC-V as shown in Figure 3.1. You will introduce all pipeline stages sequentially to maintain the functionality of the processor and to make the development easier.

3.1 First Pipeline Stage (IF/ID)

In Figure 3.2, we introduce a pipeline register between the instruction fetch (IF) and the instruction decode (ID) part of the processor. We transform the single cycle MicroRISC-V to a two stage pipelined processor, that has an instruction fetch and Decode/Execute/Writeback stage.

In particular, we add a register for the read instruction from the instruction memory, as well as for the current program counter. To make the implementation, more readable,

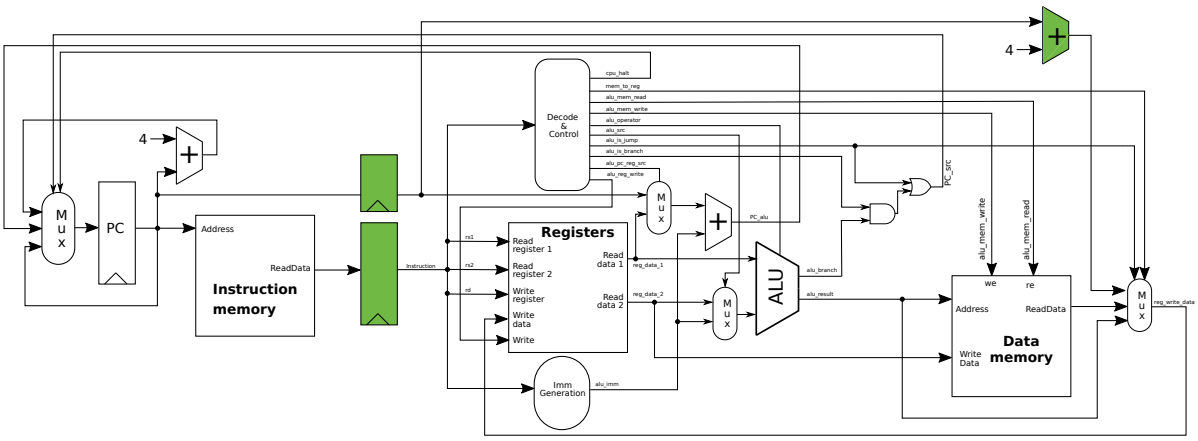

Figure 3.2: 2-stage pipelined processor.

use a special naming scheme for all pipelined registers! For example, use the suffix `_id_p`, for a pipeline register between the IF/ID stage, which is valid in the ID stage.

3.1.1 Control-Flow Hazard

A control-flow hazard occurs when a conditional branch or jump is taken in the Decode/Execute/Writeback stage. As the previous pipeline stage (IF) is assuming the next instruction to be fetched, the wrong instruction will be fetched on a taken control-flow transfer. An executed control-flow transfer in the Decode/Execute/Writeback stage is indicated when the signal `PC_src=1`.

To deal with a control-flow hazard, the execution stage needs to invalidate or stall the next instruction of the IF stage. Hint: Use dedicated pipelined 'no-operation' signal to inform the decoder to not execute the next instruction.

3.2 Second Pipeline Stage

In Figure 3.3, we introduce a pipeline register between the instruction decode (ID) and execution (EX) part of the processor.

As shown in Figure 3.3, pipeline registers are inserted for all control signals of the instruction and operation decoder, as well as for the read register values. Generally speaking, insert a pipeline register for all signals used in the later stage of the processor.

3.2.1 Control-Flow Hazard

After inserting the second pipeline stage, a control-flow hazard from the last stage, *i.e.*, the signal `PC_src=1`, influences the IF stage (as handled from the implementation of the first pipeline stage), but also the ID stage. Thus, when this signal goes high, also the current instruction in the ID stage needs to be flushed or invalidated. You can do

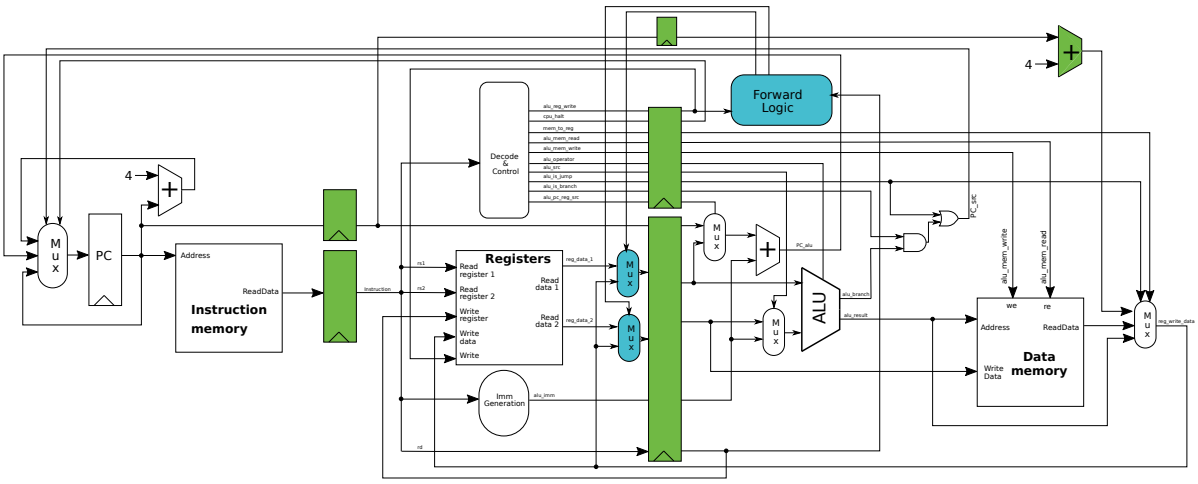

Figure 3.3: 3-stage pipelined processor.

this by implementing a *load-enable*, *i.e.*, clear all control signals of the register update if `PC_src=1` or a 'no-operation' is coming from the previous pipeline stage.

3.2.2 Data Hazards and Forwarding

When introducing the second pipeline stage, also data hazards occur. Consider the following instruction sequence, where we have an add instruction followed immediately by a subtract instruction that uses that sum (x2).

```
SUB x2, x1, x3      # Register 2 written by sub
AND x12, x2, x5     # 1st operand(x2) depends on sub
```

The second operation depends on the result of the subtraction of the first instruction. However, in the 3-stage pipeline, the result of the subtraction is not yet written to the register file when the second instruction needs it.

To solve this problem, there are two options available. The first solution is simple *stalling* the pipeline until the data is available in the register file. Of course, this reduces the performance of the processor. A better solution to this problem is so-called *forwarding*, which we implement in this assignment. With forwarding, we forward a computed result of the execution unit, in our case the signal `reg_write_data` to the input of the ID/EX pipeline stage. Thus, a new multiplexer (in blue in Figure 3.3) is inserted for the data of register port 1 and a second one for the register port 2. We now need to determine, when to forward a result rather than using the value stored in the register file. Thus, we implement a forwarding logic, which is controlling the two forwarding multiplexers. When the execute stage writes a register that is not `x0` and that registers is also a source operand of the instruction of the ID stage, forwarding needs to be active. In this case, the forwarding logic controls the forwarding multiplexer to directly use the `reg_write_data` as the register input. Since RISC-V can have two source

operands, forwarding needs to be implemented for both. Summarized, we formalize the two conditions for forwarding the result to one of the pipeline registers as follows:

1. `ID/EX.reg_write_mem != 0 && ID/EX.rd != 0 && (ID/EX.rd == IF/ID.rs1)`
2. `ID/EX.reg_write_mem != 0 && ID/EX.rd != 0 && (ID/EX.rd == IF/ID.rs2)`

Implement the forwarding logic with both conditions controlling the multiplexers. In the example code above, the first forwarding condition is true as the `and` instruction requires the result of `sub` instruction, stored in `x2`.

3.3 Third Pipeline Stage

In Figure 3.4, we introduce a pipeline register between the execution (EX) and memory/write-back (MEM) part of the processor. With this, MicroRISC-V transforms to a 4-stage processor.

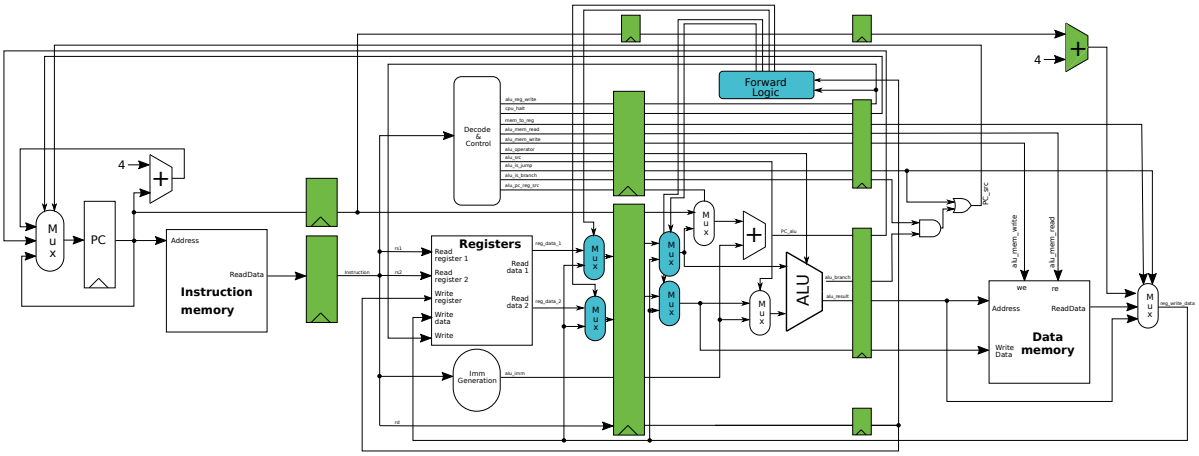

Figure 3.4: 4-stage pipelined processor.

Insert pipeline registers for all ALU-related and control signals needed in the next stage.

3.3.1 Control-Flow Hazard

A control-flow hazard also affects the new stage. Similar as before, clear all control signals of the EX/MEM pipeline stage in case of a control-flow hazard, where `PC_src=1`.

3.3.2 Data Hazards and Forwarding

Let's consider the extended code example below. Because of adding another pipeline stage before writing the result of a computation back to the register file, one more instruction can depend on the result in the execution stage.

```
SUB x2, x1, x3      # Register 2 written by sub
AND x12, x2, x5     # 1st operand(x2) depends on sub
OR  x13, x6, x2     # 2nd operand(x2) depends on sub
```

To deal with that problem, you need to extend the forwarding logic and add a second pair of multiplexers in front of the ALU, which is indicated in Figure 3.4. Similar to the first implementation of the forwarding logic, we formalize the conditions for forwarding the result to one of the pipeline stages as follows:

1. `EX/Mem.reg_write_mem && EX/MEM.rd != 0 && (EX/MEM.rd == IF/ID.rs1)`
2. `EX/Mem.reg_write_mem && EX/MEM.rd != 0 && (EX/MEM.rd == IF/ID.rs2)`
3. `EX/Mem.reg_write_mem && EX/MEM.rd != 0 && (EX/MEM.rd == ID/EX.rs1)`
4. `EX/Mem.reg_write_mem && EX/MEM.rd != 0 && (EX/MEM.rd == ID/EX.rs2)`

The first two conditions control the multiplexer of the ID stage, which was already implemented in the previous task. The second pair of conditions control the new multiplexers in the execution stage.

3.4 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 complete to give your TA an overview.
2. Submit your modified MicroRISC-V implementation including all files of the upstream repository.

3.5 Hints

1. 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.
2. Run `make view TARGET=<program-name>` to simulate your CPU and view the signal trace with GTKWave.
3. Run `make sim TARGET=<program-name>` to execute the program on the ISA simulator to observe the expected behavior.
4. Pipeline the CPU stage by stage. Start with the first pipeline stage and ensure that all tests pass before continuing the development.

5. Insert first the pipeline registers and then continue with special handling of signals, *i.e.*, dealing with hazards.

6. When introducing a register pipeline, stick with a naming scheme for all pipeline registers, e.g., use the suffix `_id_p` for a pipeline register of the IF/ID pipeline stage.

4 Task 2.b: Quicksort 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 Quicksort algorithm in RISC-V. The assignment repository contains an existing C implementation, which you modify to assembly-like instruction in C. Finally, you implement this algorithm in pure assembly and execute it on the RISC-V simulator.

4.1 Quicksort Algorithm

The quicksort algorithm takes an unsorted array as input and returns the sorted array. Therefore, the sorting algorithm applies the divide and conquer software paradigm and splits the sorting problem into several parts. Concretely, the algorithm chooses a pivot element and partitions the other elements of the given array into two sub-arrays by putting all smaller elements before the pivot and all greater elements behind the pivot element. In our case, we are going to use the last element of the array as the pivot element. Moreover, the sub-arrays are sorted recursively using the partitioning function. The Wikipedia article provides a graphical visualization of the quicksort algorithm. Furthermore, `qsort.c` contains the reference implementation of this algorithm.

4.2 Specification

Our goal is to implement the quicksort algorithm using the RISC-V assembly language. To get started, you are given a complete implementation in the C programming language. 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. Subsequently, the task is to convert the implementation to assembly. What does the program do?

`main` The main function 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`, `qsort`, and `output` in succession.

`input` first reads the number of elements being processed from stdin. Then it reads this amount of values from the stdin and stores it in the given array.

`qsort` uses the `partition` function in order to process the received array and recursively calls `qsort` using the partitioned two sub-arrays until the array is sorted.

partition is responsible for selecting a pivot element of the given array. Note that our C implementation uses the last element of the array as the pivot element. This function places the pivot element at the correct position of the array by putting all smaller elements before the pivot and all greater elements behind the pivot element.

swap exchanges the values of parameters `x` and `y` and is used by the `partition` function.

output prints the sorted array to stdout.

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

qsort.c This file provides you a complete C implementation. Use this file to understand the implementation.

qsort_transformed.c In this file, the functions `qsort`, `partition`, and `swap` are not implemented. It is your task to implement these functions in such a way that each line in `qsort_transformed.c` corresponds to precisely one instruction in `qsort.asm`. This holds true for all lines except for the function entries and exits, which lead to corresponding prologues and epilogues in assembly. Note, the function `qsort` 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 example references.

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

qsort.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 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 complete to give your TA an overview.
2. Modify `qsort_transformed.c` to provide your transformed implementation of the functions `qsort`, `partition`, and `swap`.
3. Modify `qsort.asm` to provide your assembly implementation of the functions `qsort`, `partition`, and `swap`.

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 0.7.

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: Ethernet Switching

Ethernet is a key technology of Local Area Networks. In this layer, we receive *frames* of data from one ethernet port, then need to decide where to forward them to based on the MAC address. Your task is to implement an abstract Ethernet switch in C/C++ for the specified logic. A visualization framework is provided. Be aware that submissions will be compiled as C++, and you can use any features of the C++ standard library, but doing so is not required to solve the task. A test suite is not provided, since it should suffice to think through which segments should reach which connected machines.

5.1 Setup

Compile the provided stub implementation using `make`, and start it up using `make run`. You can now access the visualization framework at `http://localhost:13354/`.

You can try sending Ethernet frames between the connected computers using the control panel on the right. To simulate a frame, left-click the button in the table row corresponding to the frame's origin, and in the column corresponding to its destination. For example, to send a frame from the machine connected to port 4 to the machine connected to port 2, click the button in the row labeled `04` and the column `02`. You can right-click the same button to simulate a frame with an incorrect checksum. To simply generate a constant flow of packets, click on "auto-generate data".

However, the switch won't properly forward segments yet – this is where *you* come in!

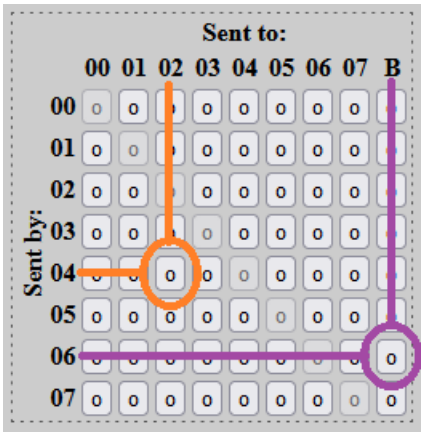

Figure 5.1: The orange button sends a frame from port `04` to port `02`.
The purple button sends a frame from port `06` to the broadcast address.

5.2 Your task

Open up `switch.h` and `switch.cpp`. To clear this task, you will need to implement `EthernetSwitch::processFrames` in `switch.cpp`. This function should:

- Retrieve any pending frames from the connections
  - Check if a frame is pending using `EthernetPort::hasFrame()`
  - If yes, retrieve the frame using `EthernetPort::getFrame()`
  - The provided method stub already does this for you
- Decide if the frames are undamaged by verifying the CRC32 checksum
  - You should use the `CalculateFrameChecksum` method for this
- Forward undamaged frames to the appropriate switch port(s)
  - To send a frame out of a port, use `EthernetPort::queueSend(frame)`
  - Frames should only be forwarded out of the appropriate port, if known
    - If the appropriate port is unknown, frames may be sent to all ports

Furthermore, you need to support very basic Virtual LAN segmentation. Virtual LANs are used to separate a physical network into smaller, isolated sub-networks (e.g., departments within one university building). To this effect, implement the function `EthernetSwitch::setPortVLAN` in `switch.cpp`, which should change the VLAN ID assigned to a specific port.

- Each port can only be in a single VLAN
- By default, all ports are in a VLAN with ID `0`
- `EthernetSwitch::setPortVLAN` updates a port's VLAN ID
- A packet received on a port belonging to VLAN n must not be sent out of a port belonging to VLAN m, for $n \neq m$.
  - Unicast packets must not be forwarded to a destination host in a different VLAN, and should be flooded to all ports in the source VLAN instead
  - Broadcast packets should be flooded only to ports in the source VLAN

Any frames transmitted from a port in one VLAN should only ever be received by devices in that same VLAN. This includes both unicast frames and broadcast frames.

5.3 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`.

- Push the final versions to your git repository on GitLab
- Remember to create and push a tag as described in Section 0.7

Your submission should run without crashing or leaking memory. Use the Valgrind-enabled[1] target (`make valgrind`) in the provided `Makefile` and test your program thoroughly!

[1] You may need to install Valgrind using `sudo apt-get install valgrind`, first.

6 Task 3.b: A tiny HTTP Server

The **H**yper**T**ext **T**ransfer **P**rotocol is the foundation of the modern internet. Every web page you visit is the result of your browser making repeated HTTP requests for content.

In this task, you will implement a basic HTTP server in C/C++. Submissions will be compiled as C++, and you can use any features of the C++ standard library. Doing so is not required to solve the task. You have to accept connections from a web browser, and respond to its requests using a minimal HTTP feature set. When you're done, your server will be able to serve up a static web site.

6.1 Setup

Your repository contains the files `http.cpp` and `server_framework.cpp`.

- `server_framework.cpp`, provided by us, sets up a TCP server listening port `8000` and accepts any incoming connection.
- Implement `handle_connection` in `http.cpp` to handle this connection.

To run the server, use `make run`. You can access it using your web browser, at `http://localhost:8000` – however, it won't actually react to HTTP requests yet. This is where you come in.

6.2 The Hypertext Transfer Protocol

The client sends a HTTP request on the connection, which you can retrieve using `read`[1]. Pay attention to `read`'s behavior – it will return once *any* data has arrived. You will not necessarily get the entire HTTP request in a single call to `read`. If you are looking for a line break, and you can't find it, call `read` again to wait for more data from the connection.

In both requests and responses, all lines should be terminated with `\r\n`. You are allowed to (but do not have to) also accept termination with only `\n` (without `\r`) in incoming requests. Your responses *must* use `\r\n` as specified by the HTTP standard.

A HTTP request starts with a *request line* of form *`method`*␣*`path`*␣`HTTP/1.1`[2]. You are required to implement the `GET` and `HEAD` methods in your server. Any other method should be met with an error message – see Section 6.2.1. *`path`* specifies the path of

[1] see man read(2)

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

the requested file, and must start with a leading slash. As a special case, you should treat any path with a trailing slash as a request for `index.html` in that directory. For example, `GET /foo/ HTTP/1.1` should be treated as a request for `/foo/index.html`.

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. For the base task 3b, you only need to verify that a non-empty `Host` header is present, and reject the request if this is not the case.

Following the header, a client request might usually include a request body – for example, data to be uploaded to the server. As your supported methods – `GET` and `HEAD` – do not allow a request body, any content after the header can be ignored. You are allowed to (but do not have to) reject the request with an appropriate error message if a request body is present – see Section 6.2.1.

Once you have processed the client's request, you should open the requested file[4] from the `webroot` directory. For example, if the path `/img/kitten.jpg` is requested, you should open `webroot/img/kitten.jpg`.[5] Ensure that the path specifies a *regular file*, and not, e.g., a directory[6]. If it does not, or the specified file does not exist, send an appropriate error message with status code 404 – see Section 6.2.1 for details.

Otherwise, respond[7] with a `HTTP/1.1␣200␣OK` status line, followed by the `Connection` header set to `close`, and the `Content-Length` header set to a decimal representation of the file's size in bytes. Terminate the header with an empty line. If the request method was `GET`, write the file's contents to the connection. Keep in mind that the the file's size might exceed the size of your server's memory! Afterwards, close[8] the connection.

6.2.1 HTTP error codes

If an error condition occurred, you should respond with an appropriate HTTP status code to indicate this to the user. To do so, write a `HTTP/1.1` *`status_code status_text`* status line, followed by an empty line indicating no headers, then close the connection.

The following are some HTTP status codes that are relevant to your work:

- If the request method is not `GET` or `HEAD`, you must respond with `501 Method Not Implemented`.
- If a request for a non-existant file is made, you must respond with `404 Not Found`.

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

[4] hint: use `We recommend fopen(3)` in C, or `std::ifstream` in C++; this is not mandatory, you may use whatever means of reading the file you want

[5] We may test your implementation with files different from the ones in the example folder. Your implementation has to work with arbitrary paths, including files in sub-folders.

[6] You can use `fstat(2)` and `S_ISREG` in C, or `std::filesystem::is_regular_file` in C++

[7] We recommend `write(2)` or `dprintf(3)`

[8] `man close(2)`

- If the client keeps sends an unreasonable amount of data before the first terminating newline, you should choose a suitable cut-off[9] beyond which to discard the request and respond with `414 URI Too Long`. This prevents a malfunctioning or malicious client from consuming an excessive amount of server resources.
- Similarly, if the client keeps sending an unreasonable amount of data before terminating the header, you should choose to discard the request and respond with `431 Request Header Fields Too Large`.
- If you discard the client's request for a reason for which no suitable status code exists, or if you do not wish to disclose the reason, you may respond with `400 Bad Request`. In particular, if a client request does not include a non-empty `Host` header, you *must* respond with `400 Bad Request`.
- If something has gone wrong on the server – for example, a file read operation has failed spuriously – respond with `500 Internal Server Error` You can use `500` as a general fall-back for any unexpected mishap.

6.2.2 Content-Type header

The `Content-Type` response header informs the client of the requested document's MIME type. This is a standardized text string identifying a particular file type.

Adjust your implementation to return a `Content-Type` header on successful requests. To determine a file's type, you may simply check its file extension on disk. The following file extensions and corresponding MIME types must be supported:

- `.html` files with MIME type `text/html`
- `.js` files with MIME type `text/javascript`
- `.css` files with MIME type `text/css`
- `.jpg` files with MIME type `image/jpeg`
- `.ico` files with MIME type `image/x-icon`
- `.txt` files with MIME type `text/plain`

You are allowed to support any other MIME types if you wish, but only the six types above must be supported to clear the task. If a requested file does not have a file extension you recognize, you must not return any `Content-Type` header.

[9] Do not choose a cut-off of ≤ 1024 bytes for request line and headers combined, as our test system may make requests of this size

6.2.3 Prevent webroot escape attacks

On the internet, not everyone will be playing nice. What would your server do if, instead of sending a request for `/img/kitten.jpg`, the client instead sends a request for `/../../../etc/passwd`? You probably don't want it sending a sensitive file to anyone that asks!

For this task, you will add a check that ensures only files in the `webroot` directory, or its sub-folders, can be requested. For example, if you are using C, you can use `realpath`[10] to resolve the requested path, and ensure it is a sub-folder of `realpath("webroot")`. If you are using C++, you could instead use `std::filesystem::canonical`, and compare the paths using `std::mismatch`. These are only two possible solutions – if you think you've come up with another one that works for you, go right ahead!

6.3 Getting started

Your task may seem daunting, so here are some guardrails you can follow on your first few steps, if you'd like.

Start out by calling `read` on the connection. It returns the number of bytes it received. Scan those bytes and see if you received a line break. If you did, everything before the line break is the request line. Split it into its components as described in Section 6.2. If you didn't receive a line break yet, call `read` again until you do.

Once you're done with the request line, repeat the process. Make sure you don't forget about any data you already received after that first line break! Once you find another one, that's your first header line. Break it into pieces and deal with it.

Repeat the process until you find an empty line. Now you're done reading the request, and just need to deal with the pieces appropriately. Good luck!

6.4 Bonus: Implement range requests

If you reach at least 10 points (*i.e.*, half) on the previous parts of Task 3.b, you are eligible for this bonus task. If you implement it, you can get additional 4 points. Thus, you can reach up to 24 points for this exercise and 104 points in the practicals.

Often times, a client might only want certain parts of a file, instead of the whole thing. For example, the user might be skipping to certain parts of a hour-long video file, or a previously-interrupted download might be resumed.

If the server indicates support using the `Accept-Ranges` header, the client may request only specific parts of a file. To clear this bonus task, you will implement a limited subset of the range request specification.[11]

You should *always* include a `Accept-Ranges: bytes` header on *any* successful request, including non-range requests. When handling a request, check whether a `Range` header

[10] man realpath(3)

[11] Note that we do not require you to support multipart ranges.

is present. If it is, it should have one of the following forms:[12]

- `bytes=`*`start`*`-` requests all bytes from index[13] *start* to the end of the file.
- `bytes=`*`start`*`-`*`end`* requests all bytes from the *start*th to the *end*th byte of the file. (The *end*th byte must be included!)
- `bytes=-`*`suffix`* requests the last *suffix* bytes at the end of the file. If the file has fewer than *suffix* bytes, requests the entire file.

If the `Range` header is present, but does not have this form, discard the request with `400 Bad Request`. If the range header is valid, but *start* is out of bounds, *i.e.*, not less than the size of the file, respond with `416 Range Not Satisfiable` and specify a `Content-Range: bytes */`*`size`* header, where *size* is the size of the requested file in bytes. If *start* is greater than *end*, do the same.

Otherwise, respond with `206 Partial Content`, and specify a `Content-Range` header. This header should take the form `Content-Range: bytes `*`start`*`-`*`end`*`/`*`total`*, specifying the *start* and *end* index of the range you sent, as well as the *total* size of the file. In the response body, send only the requested byte range from the file, and set the number of bytes in the range for the `Content-Length` header. If *end* is out of bounds, but *start* is not, truncate the range by setting `end` to the maximum value possible, and send the truncated range in the `Content-Range` header. See Section 6.6.5 for an example.

All other response headers should be sent as usual.

6.5 Deliverables

All files must be submitted in folder `task-3b` of your repository. You may freely modify any files in this directory, or add files to this directory. The test system will process all `*.cpp` files in this directory.

- Push the final versions to your git repository on GitLab
- Remember to create and push a tag as described in Section 0.7

Your submission should run without crashing or leaking memory, including on unexpected input. Use the Valgrind-enabled[14] target (`make valgrind`) in the provided `Makefile`, and test your program thoroughly. Consider what malformed inputs you might encounter, and make sure you reject them gracefully!

[12] No whitespace within the header's value is permitted. Whitespace between `Range:` and the header value must be discarded as described in Section 6.2.

[13] The first byte of the file has index zero; valid indices in a file of size n range from 0 to $n - 1$.

[14] You may need to install Valgrind using `sudo apt-get install valgrind`, first.

6.6 Example HTTP requests and responses

This section contains some HTTP request + response pairs for demonstration purposes. Note that this is *not* intended to be a complete list of test cases for your implementation.

6.6.1 Basic HTTP request

```
GET /lorem.txt HTTP/1.1
Host: localhost:12345
```

```
HTTP/1.1 200 OK
Connection: close
Content-Type: text/plain
Content-Length: 17

lorem ipsum dolor
```

6.6.2 Basic error response

```
GET /iorem.txt HTTP/1.1
Host: localhost:12345
```

```
HTTP/1.1 404 Not Found
```

6.6.3 Request for a subfolder's index document

```
GET /foo/ HTTP/1.1
Host: localhost:12345
```

This should be interpreted as a request for `/foo/index.html`, as the path has a trailing slash. Note that `GET /foo HTTP/1.1`, without a trailing slash, should result in a `404 Not Found`, as `webroot/foo` is not a *regular file*.

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 50
Content-Type: text/html

<html><head><title>Foo index</title></head></html>
```

6.6.4 Fully satisfiable range request (*start* $\leq$ *end* $<$ *total*)

```
GET /lorem.txt HTTP/1.1
Host: localhost:12345
Range: bytes=4-9
```

Note that the first byte has index 0, as shown below:

l	o	r	e	m	␣	i	p	s	u	m	␣	d	o	l	o	r
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

```
HTTP/1.1 206 Partial Content
Accept-Ranges: bytes
Connection: close
Content-Length: 6
Content-Range: bytes 4-9/17

m ipsu
```

6.6.5 Partially satisfiable range request (*start* $<$ *total* $\leq$ *end*)

```
GET /lorem.txt HTTP/1.1
Host: localhost:12345
Range: bytes=10-19
```

```
HTTP/1.1 206 Partial Content
Accept-Ranges: bytes
Connection: close
Content-Length: 7
Content-Range: bytes 10-16/17

m dolor
```

6.6.6 Out-of-bounds range request (*total* $\leq$ *start*)

```
GET /lorem.txt HTTP/1.1
Host: localhost:12345
Range: bytes=20-29
```

```
HTTP/1.1 416 Range Not Satisfiable
Content-Range: bytes */17
```

6.6.7 Suffix range request 1

```
GET /lorem.txt HTTP/1.1
Host: localhost:12345
Range: bytes=8-
```

```
HTTP/1.1 206 Partial Content
Accept-Ranges: bytes
Connection: close
Content-Length: 9
Content-Range: bytes 8-16/17

sum dolor
```

6.6.8 Suffix range request 2

```
GET /lorem.txt HTTP/1.1
Host: localhost:12345
Range: bytes=-8
```

```
HTTP/1.1 206 Partial Content
Accept-Ranges: bytes
Connection: close
Content-Length: 8
Content-Range: bytes 9-16/17

um dolor
```

7 Errata

This chapter lists releases and changes of this file.

2021-10-08 Initial release.

2021-10-09 Correct the ASM diagram of Task 1a.

2021-11-05 dividend/divisor bug of Task 1b fixed. Editorial changes for Task 2a.

2021-12-03 Task 3 published.