Computer Organization and Networks Practicals 2021/22

November 5, 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.

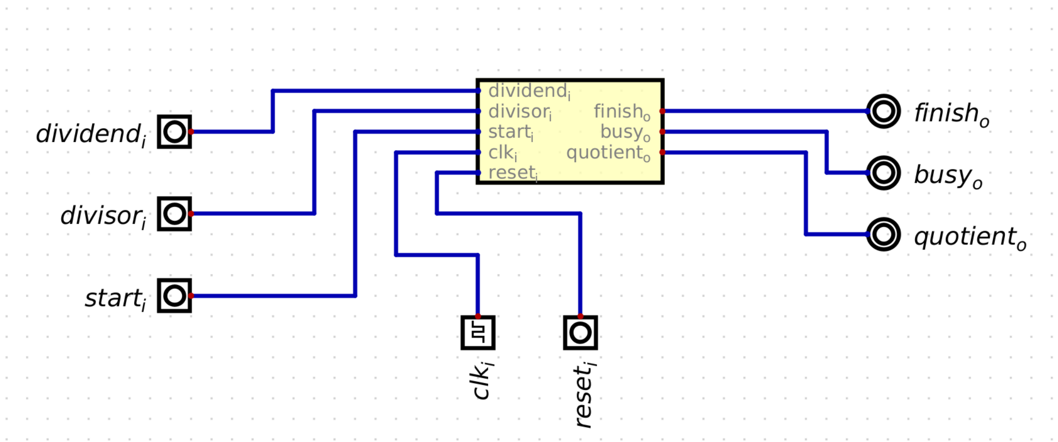

Figure 1.1: High-level view of the Divider in Digital.

```
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.

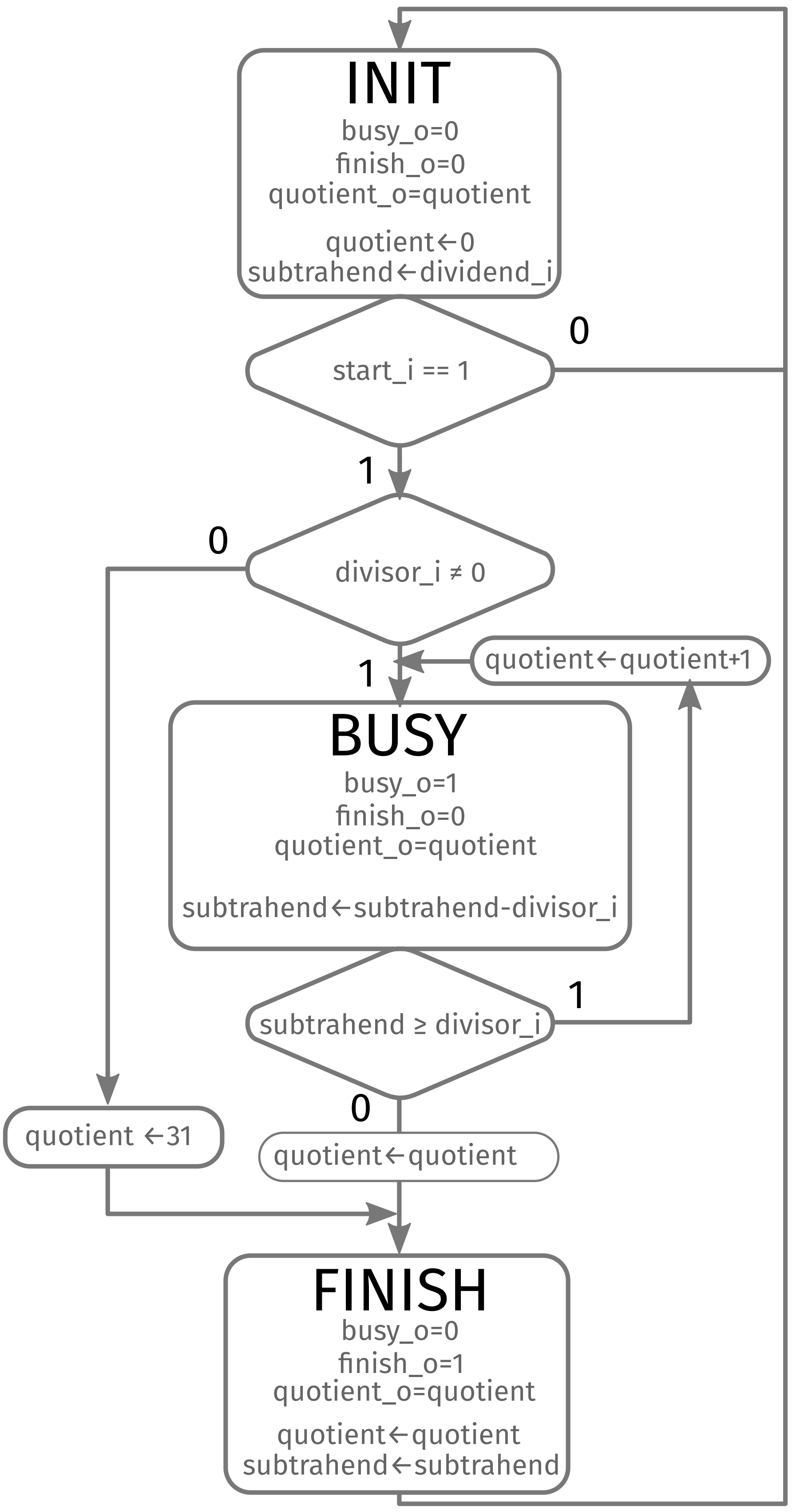

Figure 1.3: ASM diagram of the divider.

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 a plagiarism.

   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 divisor 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 divisor 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

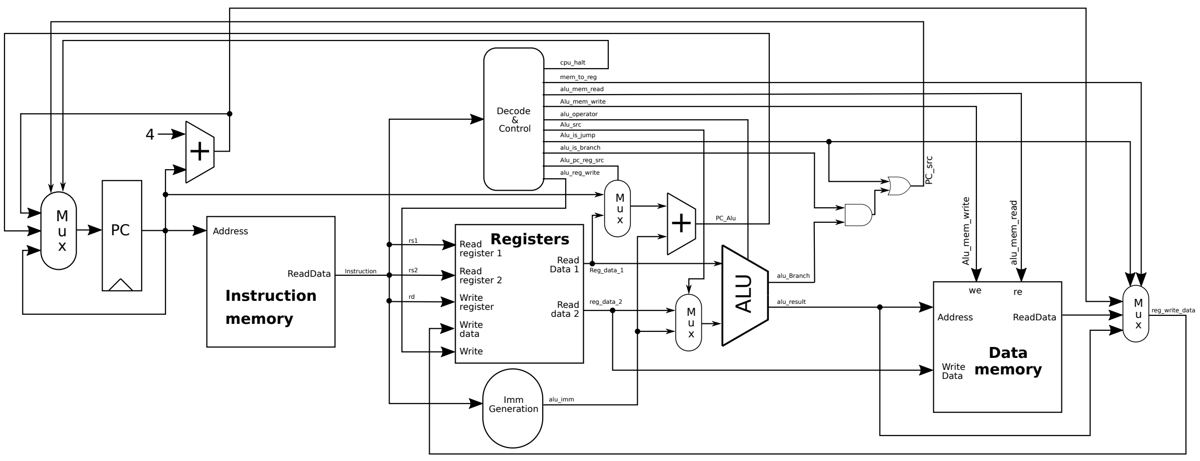

Figure 2.1: MicroRISC-V architecture.

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.

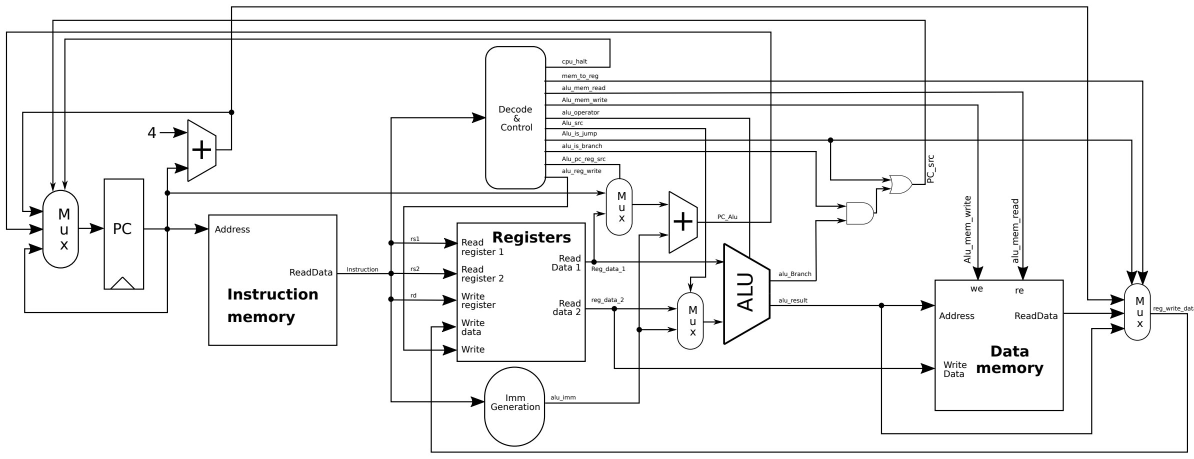

Figure 3.1: Starting Point: Single-cycle MicroRISC-V CPU.

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,

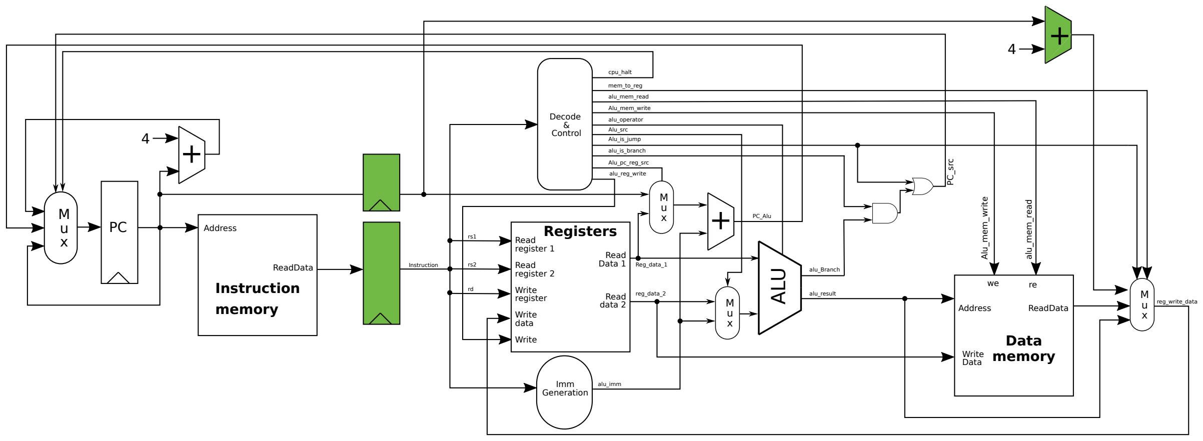

Figure 3.2: 2-stage pipelined processor.

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

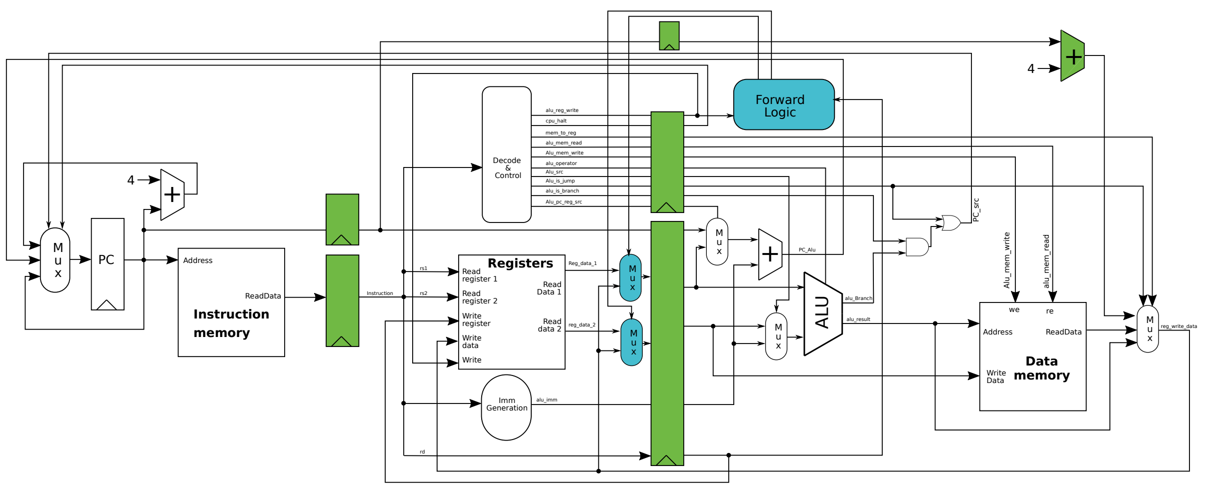

Figure 3.3: 3-stage pipelined processor.

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.

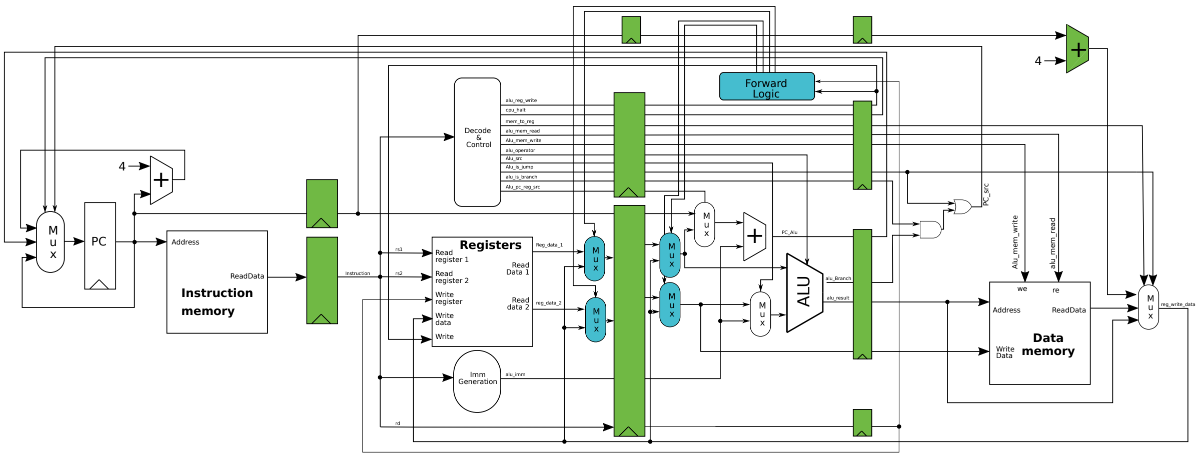

Figure 3.4: 4-stage pipelined processor.

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 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 FIx dividend/divisor