Dipl.-Ing. Dr. Tobias Scheipel, BSc

David Beikircher, BSc
Florian Riedl, BSc

448.029

Microcontroller Design, Laboratory

Instruction Guide

HADES-V

Embedded Architectures & Systems Group
Institute of Technical Informatics
Graz University of Technology

HADES : *The Greek underworld, in mythology, is an otherworld where souls go after death, and is the original Greek idea of afterlife. At the moment of death the soul is separated from the corpse, taking on the shape of the former person, and is transported to the entrance of the Underworld [by Charon across the river Styx, see Figure 1]. The Underworld itself is described as being either at the outer bounds of the ocean or beneath the depths or ends of the earth. It is considered the dark counterpart to the brightness of Mount Olympus, and is the kingdom of the dead that corresponds to the kingdom of the gods.* HADES *is a realm invisible to the living, made solely for the dead.*

– Wikipedia: The Free Encyclopedia. Wikimedia Foundation, Inc. Web. 09 Feb 2017.

Figure 1: *Crossing the River Styx* by Joachim Patinir (1480-1524), Prado Museum, Madrid.

In this course, HADES is simply for "Hardware Design" and HADES-V refers to a RISC-V processor to be designed throughout the course.

This course builds on the rich legacy of the HaDes course, which has been a cornerstone of processor design education for over two decades. Originally developed at the University of Würzburg under Prof. Reiner Kolla's and Marcel Baunach's guidance, HaDes became widely recognized for its innovative teaching approach and the Best Processor Award, an initiative celebrating outstanding student-designed processors. Many of these projects set benchmarks in creativity and technical achievement, leaving a lasting impact on generations of students.

Following Prof. Baunach's appointment at Graz University of Technology, the course continued to evolve, carrying forward its tradition of excellence in processor design education. For this Open Educational Resource, HADES-V has been fully reimagined and redesigned to feature a modular, pipelined RISC-V processor while sticking to its roots. This transformation embraces the principles of open hardware and modern processor architectures, making the course accessible to a global audience while preserving its strong educational foundation.

Version: 2024-12-17

Contents

1 Prologue

THE MICROCONTROLLER DESIGN, LAB is about deeply understanding the internal concepts, structure, and operation of Microcontroller Units (MCUs). Throughout this course, you must implement a specific RISC-V-based [1, 2] MCU in a Hardware Description Language (HDL) and write software for it in *assembly* and *C*. You will have honed your skills in describing digital logic, implementing low-level code, and debugging software and hardware using various tools. The exercises will be done in SystemVerilog on an AMD Field-Programmable Gate Array (FPGA), provided on the Basys3 [3] board from Digilent, as depicted in Figure 1.1.

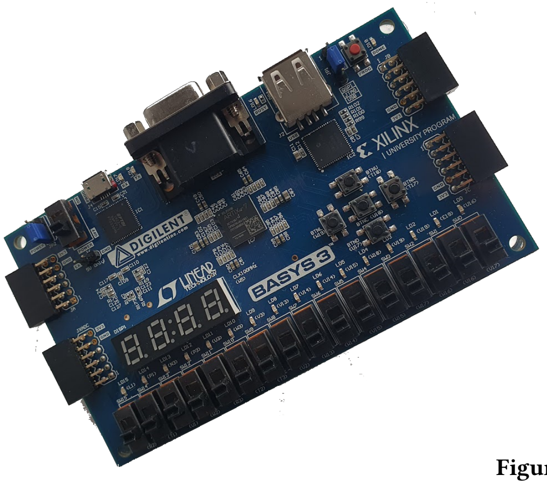
Figure 1.1: The Basys3 board featuring an FPGA.

WE WANT YOU TO BE SUCCESSFUL! The Microcontroller Design, Lab takes place as a standalone laboratory. Nevertheless, it requires a sound understanding of a broad spectrum of topics *and* the ability to link this knowledge well-structured and creatively. Even though there are no formal requirements on previously passed courses, it is highly advisable to have some knowledge about "processor architectures" and "hardware description languages".

HOW DOES THE LAB WORK? A lab session (starting at 9am) consists of two parts: At first, a presentation session with **compulsory** attendance is scheduled, where some participants present their implementation to the others. This allows you to look at other implementations and discuss the implementation decisions. The presentation is part of your grade! Presentation sessions are named after an exercise and will **only** take place if an exercise is due this week (check lab schedule!). The second part (attendance voluntary) is a supervised lab, allowing asking the supervisors about your implementation troubles.

In any case, each participant must complete the exercises described in Chapter 4 individually within the given deadlines. The exercises can be completed at home or in the lab, while the time in the lab offers the opportunity for questions and discussions. For remote (unsupervised) communication, please refer to

`https://matrix.to/#/#mdlab:chat.tugraz.at`.

You will be evaluated! Each participant is evaluated individually. Therefore, we will review your source code regarding completeness, functionality, structure, and documentation/comments. To do so, we use an automatic test system that evaluates your submission. Also, readability and code quality influence your evaluation. You need at least 1 point for per submission for a positive grade. The points you get on your submissions make 75% of your final grade.

Furthermore, we will evaluate the presentation of your implementation to other students, which takes place during the presentation sessions. You must also be able to answer questions (from the supervisor and from your colleagues) related to the current exercise and your implementation. The quality of the answers and the participation will influence your final grade. The average points you get on all your presentations make up 25% of your final grade.

A final word! Passing the Microcontroller Design, Lab will require a significant amount of time. Nevertheless, the content will give you a good understanding of the functionality of a microcontroller.

Have Fun and Success!

2 Toolchain Preparation

In this chapter, we will guide you through the toolchain installation process for a Linux system and shortly present the used tools[1]. Please only try to install something after reading the rest of this chapter! Some details here are essential for the correct installation. Before you start with the implementation, please restart your computer[2].

1 Note that a virtual machine will also be provided with all the necessary tools pre-installed. You can use this virtual machine on the lab PCs or download it for personal use.

2 Also, you will find margin notes like this along the way. This way, your eye catches essential information more easily.

2.1 SystemVerilog

SystemVerilog [4] is an advanced Hardware Description Language (HDL) for designing, verifying, and modeling digital systems. It has evolved from Verilog and improves design capabilities by providing features for modeling and verifying complex systems. This language combines hardware description, constraints, and test mechanisms, streamlining the creation and verification of complex designs. With its object-oriented programming capabilities, SystemVerilog enables scalable and reusable designs to build complex hardware systems efficiently.

In this course, we presume knowledge of SystemVerilog[3] and ask you to structure your code and it is recommended to follow the coding style presented in Appendix B, as your coding style is also part of the evaluation.

3 If you are unfamiliar with SystemVerilog, the book by Donald Thomas [5] provides a good introduction.

2.2 Verilator

Verilator[4] is an open-source software tool for converting SystemVerilog and Verilog HDL code into a cycle-accurate *C++* or *SystemC* behavioral model. This model offers higher performance than event-driven simulators, which model behavior within the clock cycle. It acts as a fast simulator that transforms HDL code into an executable form for simulation and testing.

4 The official website of Verilator is: `https://verilator.org`

To obtain the correct version of Verilator, it has to be built from source by running the following commands[5]:

```
sudo apt install git autoconf g++ libfl-dev help2man
git clone --branch v5.006 https://github.com/verilator/verilator
cd verilator
autoconf
./configure
make
sudo make install
```

5 Depending on your hardware and internet connection, this step may take some time ($\sim$ 30 minutes).

2.3 RISC-V Toolchain

A version of **gcc** targeting **rv32i** without extensions needs to be compiled from source code. This is done by executing the following commands[6]:

```
sudo apt install texinfo zlib1g-dev libexpat-dev libgmp-dev
git clone --branch 2023.01.31 https://github.com/riscv-collab/riscv-gnu-toolchain.git
cd riscv-gnu-toolchain
./configure --with-arch=rv32i --prefix=/opt/riscv32i
sudo make
```

6 Depending on your hardware and internet connection, this step may take very long (~ 4 hours). Also, be aware that there may be no indication of progress for long periods.

2.4 AMD Vivado

Vivado™ is a synthesis tool that generates a netlist for the FPGA. It is offered by AMD in different editions. The WebPACK edition is free and sufficient for our purposes. You can find the download at `http://www.xilinx.com/support/download.html`. To get the installer[7], you must register for free at the AMD homepage. Vivado™ can also be used for developing and simulating your HDL codes. During installation, choose WebPACK. Get the *Free SDK, Vivado™ WebPACK*, and press “Connect Now”. A browser opens and lets you select and activate the free available licenses.
For newer Vivado™, generating additional locales may be necessary:

```
locale-gen "en_US.UTF-8"
```

7 The minimum Vivado™ version is 2019.2, but newer versions work as well.

Without ncurses, the installer will get stuck when generating some files. Therefore, it needs to be installed first:

```
sudo apt install libncurses5
```

Now extract the `.tar.gz` file using the archive manager. Then, launch the **xsetup** file in the extracted directory as root (`sudo`) and follow the installation instructions. Make sure to select `Vivado HL WebPACK`. Disable all optional features except for **Artix-7** support.

Finally, install the cable drivers:

```
cd /opt/Xilinx/Vivado/<VERSION>/data/xicom/cable_drivers/lin64/install_script/install_drivers
sudo ./install_drivers
```

2.5 GTKWave

Verilator compiles and executes code on your host system and outputs the result textually. The textual form is helpful for information, warnings, errors, and assertion messages. However, analyzing concurrent signals in a so-called “wave viewer” is often beneficial. A widely used open-source wave viewer is GTKWave[8].

8 You can download the viewer from `http://gtkwave.sourceforge.net/` (or by executing `sudo apt install gtkwave`).

2.6 GIT

GIT is an open-source version control system. We will use it to provide the development environment, and you will use it to upload your code. You can use any GIT client you prefer (e.g., `https://git-scm.com/`).

3 Getting Ready

After a successful toolchain installation or virtual machine image import, you can download our prepared development environment from our GIT server. We have already created a template to use for your own GIT repository. This chapter presents step-by-step instructions on how to download the environment.

3.1 How to Create a GIT Repository?

Open the URL

`https://gitlab.tugraz.at/`

and sign in using your TUGRAZOnline username and password. Create a new empty[1] GIT repository according to the following naming rules[2]:

```
MDLab_<SURNAME>_<REGNUMBER>
```

In your repo's settings, navigate to "Members" and make sure to add your course supervisors as "Maintainer". Have a look at the TeachCenter or TUGRAZOnline if you do not know their mail addresses.

3.2 How to Checkout the Environment?

First, create and navigate to the folder to which the repository shall be downloaded. Executing the command

```
git clone <LINK_TO_YOUR_REPO>
```

will clone[3] the repository without downloading the files. This is necessary as we first must set a configuration for this repository. To do so, enter the folder by executing:

```
cd <YOUR_REPO>
```

Now, disable the automatic conversion of the line ending[4] with:

```
git config core.autocrlf false
```

Finally, add the public GIT repository[5] containing the template as upstream and pull from it:

```
git remote add upstream git@github.com:tscheipel/HaDes-V.git
git pull upstream main
git push
```

1 Do **not** initialize the repository with a README.

2 `<REGNUMBER>` is your 8-digit student registration number.

3 Set up your SSH keys beforehand, as explained here: `https://docs.gitlab.com/ee/user/ssh.html`.

4 Please do not change the line endings in the files!

5 The template is available as open source here: `https://github.com/tscheipel/HaDes-V`.

3.3 How is the Project Structure defined?

For convenient organization, configuration, and maintenance, the project files are structured in a defined way in dedicated directories. Set up your repository according to the structure shown in Figure 3.1.

Your main working directories are `rtl` and `test`. The `rtl` directory contains the HDL code of your MCU implementation, and the `test` directory includes some tests to check your implementation. Since the precompiled reference implementation is available, you can always execute the available tests when looking for errors. Note that not all edge and corner cases are tested by these tests. Hence, you must write your own tests to verify your implementation since passing the available tests does not guarantee a correct implementation.

```
*<PATH_TO_YOUR_REPO>*
├── build
│   └── *build directory*
├── defines
│   └── *type and constant definitions*
├── lib
│   └── *common hardware modules*
├── ref
│   └── *precompiled reference*
├── rtl
│   └── *your implementation*
├── saves
│   └── *gtk save files*
├── sim
│   └── *simulation top directory*
├── std
│   └── *C helper libraries*
├── synth
│   └── *synthesis top directory*
├── test
│   ├── asm
│   │   └── *assembler tests*
│   ├── c
│   │   └── *C tests*
│   └── sv
│       └── *SystemVerilog tests*
└── Makefile
```

Figure 3.1: Project folder structure. **Do NOT change this structure!**

3.4 How to Upload Files to GIT?

To let us review and evaluate your progress, you must add, commit, and push your code via GIT. If your GIT is not configured yet, make sure to do so by using the commands:

```
git config --global user.email "you@student.tugraz.at"
git config --global user.name "Firstname Lastname"
```

To reduce the storage consumption and potential problems after a compilation or synthesis on a different workstation, please refrain from adding generated output files to your repository. To add files to the version control system, use the command:

```
git add <files to add>
```

The simplest way to add all files, except the generated files, is to add all files right at the beginning, i.e., when preparing your folder. To mark the current progress of your repository with a meaningful text, use the command:

```
git commit -a -m '<text>'
```

All your commits are stored locally unless you push your changes to the GIT server using:

```
git push
```

3.5 How does the Makefile work?

Within your workspace is one `Makefile` for handling the building, execution, simulation, visualization, and synthesis of your implementation. Running `make help` displays the available commands. To test your implementation, create testbenches in *SystemVerilog*, *assembly*, or *C* and place them in their respective folders within the `test/` directory. Run the test using `make test/<asm|c|sv>/<test_name>` and view the results via `make show`[6] to visualize the outcome using gtkwave.

6 The command `make show` utilizes the same saved waveform file for *C* and *assembly* tests, while *SystemVerilog* testbenches generate their own individual save file for each testbench.

3.6 *How to Synthesise the MCU?*

After successfully implementing and simulating all CPU modules[7], you may want to test the MCU (cf. Figure 3.2) on real hardware. To generate a netlist for the FPGA, you must use a synthesis tool, like Vivado™ (see Section 2.4). Running the command

```
make synthesis
```

executes the required implementation and synthesis steps using Vivado™. A bitstream is generated upon success, which can then be programmed[8] onto the FPGA board.

3.6.1 *How to Write the Bitstream to the FPGA Board?*

After generating the bitstream, you can program it onto the FPGA board using Vivado™. Therefore, open the hardware manager in Vivado™, connect the FPGA board to the PC (forward USB to the virtual machine if needed), and ensure that the cable drivers are installed (see Section 2.4). When the hardware manager is opened, click `"Open target"` → `"Auto connect"`. When clicking `"Program device"`, a window opens where the bitstream file (`build/synth/hades-v.bit`) can be selected, and when pressing `"Program"`, the bitstream is written to the FPGA. Since the memory is initialized with the bootloader it starts automatically and waits for a program.

The bitstream can also be stored on the quad Serial Peripheral Interface (SPI) flash available on the board. This allows the board to disconnect from the power supply and restore the bitstream on the next startup. This can be achieved by opening the hardware manager and connecting the board as described above. Instead of programming the device, you can right-click on the device and click `"Add Configuration Memory Device..."` opening a window where you can select the configuration memory part: `S25FL032`. Then, you are asked to select the configuration file (`build/synth/hades-v.bin`) to be stored in the flash.

3.7 *How to Upload a Software Program?*

After the bitstream is programmed onto the FPGA, the bootloader automatically starts. This is because the MCU's memory is already initialized with the bootloader code. After initializing the system, the bootloader expects an incoming binary to execute on the Universal Asynchronous Receiver Transmitter (UART) serial interface.

Executable binary `.hex` files can be created by using the following command:

```
make test/<PATH_TO_YOUR_TEST_PROGRAM>
```

When the command executes without errors, you find the compiled code in the `build/test/<...>` directory. There is the `.hex` file that can be transmitted to the bootloader using some serial communication program like `cutecom`[9].

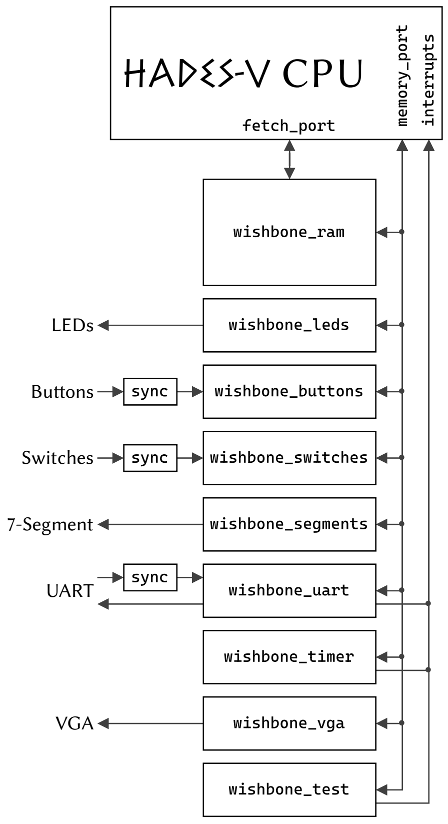

Figure 3.2: The block diagram of the HADES-V MCU, featuring the Central Processing Unit (CPU) and its peripherals.

7 Synthesis does **not** work with pre-compiled reference implementation modules instantiated in your code.

8 Ensure the synthesis raises no errors or warnings (check the `build/synth/vivado.log` file).

9 You can simply install it by executing `apt install cutecom` and start it with root permissions: `sudo cutecom`

If the board is connected and the bootloader runs, you can send the `.hex` file by clicking `“send file”`. When the program is flashed successfully, you see it on the output window, and the bootloader will jump directly to the start point of the program.

3.7.1 Bootloader

The bootloader is a special piece of software that gets loaded into Random-Access Memory (RAM) automatically whenever the CPU boots up. In our case, the bootloader is pre-programmed into the FPGA configuration file. You can find the source code for the HADES-V bootloader in `test/c/bootloader.c`[10].

Upon launch, the bootloader first copies itself to the end of RAM and sets up its own small stack (see `std/src/boot.c`). Then, it expects to receive an Intel HEX [6] file via the UART and saves it to the RAM (see `std/src/boot_internal.c`). Once the file has been downloaded successfully[11] and the checksum is verified, the bootloader jumps to the entry point of the payload program.

10 This is just a normal C program that calls the `run_bootloader` library function, so you can also call this from within your own program!

11 Since the bootloader itself also runs from RAM, the last 4kB of memory can't be used by the payload program (except for the runtime stack).

4 Introduction to the Exercises

Before we start with the exercises, this chapter will provide you with general information about the HADES-V CPU and some other important general notes about the exercises and the process within the Microcontroller Design, Lab.

4.1 HADES-V Overview

The HADES-V processor to be developed in this course is a 32-bit pipelined RISC-V[1] processor featuring five distinct pipeline stages: Instruction Fetch, Instruction Decode, Execute, Memory, and Writeback (cf. Figure 5.1). This processor is designed to execute data-dependent instructions, handle errors, and respond to interrupts.

Your implementation must support the fundamental RV32I instruction set as defined in the standard [1], which deals with basic integer operations and also all Control and Status Register (CSR) instructions [2] (referred to "Zicsr"). All the essential CSRs and also the machine timer registers are implemented. The implemented CPU supports machine mode only, focusing on efficient machine-level execution.

More information about processor architecture in conjunction with RISC-V can be found in Hennessy and Patterson [8].

1 RISC-V is an open and free Instruction Set Architecture (ISA) defined in 2010 by the University of California, Berkeley, by Waterman et al. in [1] and [2] and is inspired by established Reduced Instruction Set Computing (RISC) architectures as mentioned above. However, unlike many proprietary instruction sets, RISC-V is freely available in terms of use and modification. The ISA is designed to be easily extensible, and its standard can be implemented in different variants with different extensions. A CPU or MCU implementing the RISC-V ISA must at least contain 32 basic registers alongside some standard instructions to fulfill the minimum specification. Many different implementations exist, both hardcore and softcore CPUs and MCUs. Amongst the variants, proprietary cores can be found alongside open-source RISC-V designs in industry and academia. (cited from [7])

4.2 List of Exercises

Implement and hand in all the following exercises according to the lab schedule. Each Exercise has its own amount of maximal achievable points. When grading, functionality, automatic test case assessment, and code quality is taken into account.

Getting Ready (0 pts): Prepare your GIT repository and ensure you can run all the tools as explained in Chapter 3.

Exercise 1 (3 pts): Implement the CPU top module in `cpu.sv` according to Figure 5.1 in Chapter 5. Further general information to kickstart your course experience is given in Section 4.3.

Exercise 2 (8 pts): Implement the Instruction Fetch Stage in `fetch_stage.sv` according to the description in Section 5.1.

Exercise 3 (6 pts): Implement the Decode Stage in `decode_stage.sv` and the Register File in `register_file.sv` according to the descriptions in Section 5.2 and Section 5.2.2.

Exercise 4 (6 pts): Implement the Instruction Decoder in `instruction_decoder.sv` according to the description in Section 5.2.1.

Exercise 5 (10 pts): Implement the Execute Stage in `execute_stage.sv` according to the description in Section 5.3.

Exercise 6 (10 pts): Implement the Memory Stage in `memory_stage.sv` according to the description in Section 5.4.

Exercise 7 (16 pts): Implement the Writeback Stage in `writeback_stage.sv` according to the description in Section 5.5.

Exercise 8 (6 pts): Synthesize and test the complete MCU according to the description in Section 3.6.

Final Exercise (10 pts): Extend HADES-V in hardware or software with your own or some pre-defined ideas (see Section 4.4).

4.3 *The Start of your HADES-V Journey*

When starting off implementing the CPU top module in `cpu.sv`, have a look into the project structure first (cf. Section 3.1). Go to the **`rtl`** folder, where all the code templates are provided. You will find a code snippet that looks similar to Listing 4.1.

```
 1 module cpu (
 2     input logic clk,
 3     input logic rst,
 4 
 5     wishbone_interface.master memory_fetch_port,
 6     wishbone_interface.master memory_mem_port,
 7 
 8     input logic external_interrupt_in,
 9     input logic timer_interrupt_in
10 );
11 
12     // TODO: Delete the following line and
13     // implement this module.
14     ref_cpu golden(.*);
15 
16 endmodule
```

Listing 4.1: The template for `cpu.sv`.

This structure is similar in every template file provided in **`rtl`**. The first lines (Lines 1 to 10 in this case) always represent the module pin-out definitions.

Line 14 shows how the pre-compiled reference implementation of a module is instantiated. Please remove[2] this line and insert your code here, as the comment suggests. For the first exercise, create the top module for the CPU, as depicted in Figure 5.1.

[2] Make sure **never** to push code that features any instantiation of the pre-compiled reference implementation rather than your already implemented modules for your submission, even if they do not work correctly. It is your task to fix incorrect modules after the submission deadline, as explained in Section 4.4. Non-compliance leads to points being deducted.
Of course, not-yet-finished modules can stay as they are.

4.4 The Final Exercise

The last exercise is a free project in which the self-developed HADES-V MCU has to be extended. Hardware, software, or co-design projects are possible.

Some ideas include creating solutions for USB and PS/2, VGA text mode, Bash-like shell, or SPI, I2C, I2S, CAN, and PWM drivers. Several PMOD modules are provided that can be used to generate ideas for hardware projects, as seen in Figure 4.1.

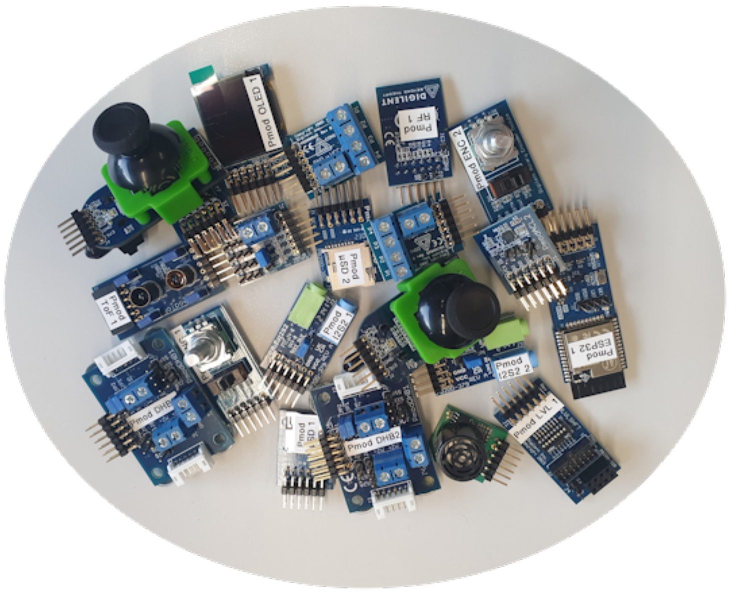

Figure 4.1: A bunch of PMOD modules to extend the HADES-V MCU.

As soon as the idea for a project is ready, the lab staff must review and confirm the plausibility and contents. The best project is evaluated during the final presentation and awarded the HADES-V Award[3].

3 The history of awardees can be viewed here: `https://iti.tugraz.at/teaching/awards/hades-award`

4.5 A Final Word on the Exercise Mode

The total achievable points sum up to a maximum of 75 pts. All exercises must be pushed onto your remote GIT repository *before* the announced exercise deadline[4]. If your submitted implementation does not work correctly, you must fix it after the deadline (ask your supervisors!). This is because the following exercises depend on your previous implementations. Follow-up errors also lead to points being deducted in the following tasks.

4 Deadline for an exercise is on the Sunday ***before*** the corresponding exercise presentation session at **23:59:59**.

5 HADES-V *Architecture*

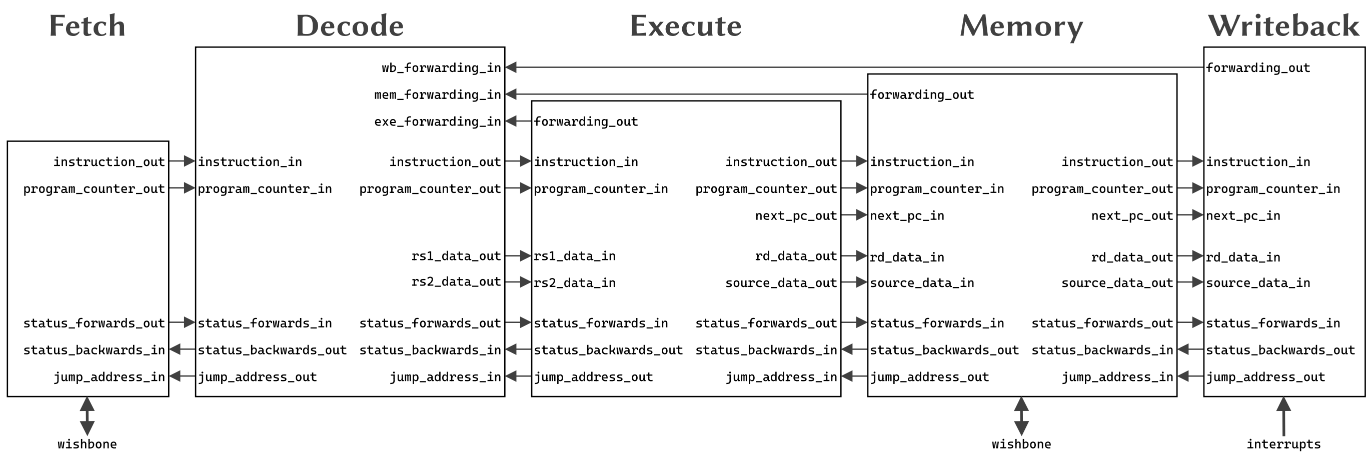

Figure 5.1: A top-level architectural overview and block diagram of the HADES-V CPU.

THROUGHOUT THE LAB, you design, implement, and test all parts of the HADES-V CPU and assemble it with all peripherals to form the HADES-V MCU. The individual parts of the CPU are explained in the present chapter.

The HADES-V CPU has a total of 5 pipeline stages (cf. Figure 5.1): An Instruction Fetch Stage, a Decode Stage, an Execute Stage, a Memory Stage, and a Writeback Stage. While each stage has its own set of features and responsibilities, some behavior[1] is shared between them (also refer to the lower three signals in the block diagram):

- Every stage must pass `status_backwards` and `jump_address` through without delay if it is a jump or a stall.
- Every stage must maintain its outputs if the following stage signals a stall (via `status_backwards`).
- Every stage must flush its own state if the following stage signals a jump (via `status_backwards`).
- Every stage must signal a stall (via `status_backwards`) if it cannot consume any inputs in the current clock cycle.
- Every stage must signal a bubble (via `status_forwards`) if it cannot provide any outputs in the current clock cycle.

1 To be consistent throughout the pipeline stages, please refer to the definitions in Sections 6.1 and 6.2.

5.1 Instruction Fetch Stage

The Instruction Fetch Stage is responsible for fetching instructions from memory. Every clock cycle, one data word is fetched from the RAM (connected via the `wishbone` interface) at the current Program Counter (PC)[2] address. The PC value must be maintained in this stage as well. This stage has the following inputs and outputs:

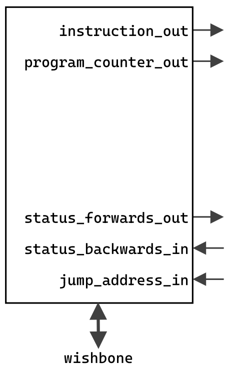

Figure 5.2: The pin-out of the Instruction Fetch Stage.

`instruction_out` provides a sequential stream of instructions fetched from RAM. The first instruction after reset must be fetched from `constants::RESET_ADDRESS`. If `status_backwards_in` indicates a jump, the next instruction is fetched from `jump_address_in`.

`program_counter_out` provides the address in memory that corresponds to `instruction_out`. If `status_forwards_out` indicates an error, this output provides the memory address corresponding to the instruction that couldn't be fetched.

`wishbone` is the interface[3] that connects to the instruction fetch port (see Figure 3.2) of the CPU. This port is *exclusively* connected to the RAM; hence, reads over this port are guaranteed to have no side effects.

[2] **Hint**: Using sequential logic for storing the current values of the stage might be helpful.

[3] For a detailed description of the wishbone signals, refer to Section 6.3.

Additionally, the Instruction Fetch Stage must respect and generate the pipeline control signals as described in Section 6.1. The only allowed error to be emitted by this stage is `FETCH_FAULT`, in case the wishbone bus signals an error.

5.2 Decode Stage

The Decode Stage decodes the instruction word from the Instruction Fetch State. It generates the necessary information to execute the instruction (e.g., set control signals and provide immediate and source register values) in the subsequent stages. Therefore, this stage contains:

- an Instruction Decoder (see Section 5.2.1) that extracts the individual information within the instruction word (e.g., operation, destination register address, source register addresses, immediate values),
- the Register File (see Section 5.2.2) that is responsible for the handling of the CPU registers (x0 – x31) [4],
- and a Forwarding Unit (see Section 6.2) that forwards results from other stages that are not stored in the Register File yet.

To make the code easier to read, the Instruction Decoder and the Register File are implemented in two separate sub-modules.

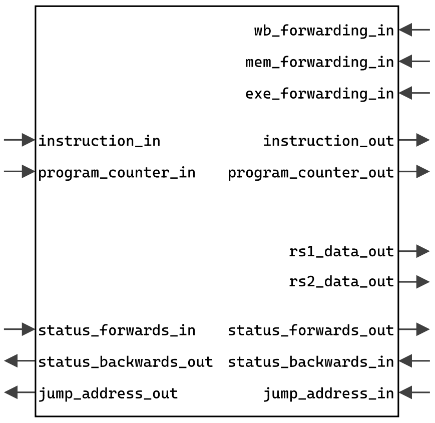

Figure 5.3: The pin-out of the Decode Stage.

4 The registers are logically written in the Writeback Stage, so you need to forward its signals to the Register File in the Decode Stage.

This stage has the following inputs and outputs:

`instruction_in` holds the instruction word fetched from the RAM.

`program_counter_in` holds the address corresponding to `instruction_in`.

`<x>_forwarding_in` hold the results of the other stages (`exe`cute, `mem`ory, `write`back) using `data_valid`, `data`, and `address` (see Section 6.2).

`instruction_out` provides the decoded instruction (see Section 5.2.1).

`program_counter_out` provides the address corresponding to `instruction_out`. If `status_forwards_out` indicates an error, this output provides the memory address corresponding to the error.

`rs<n>_data_out` provides provides register `<n>`'s source data (forward data if necessary).

Additionally, the Decode Stage must respect and generate the pipeline control signals as described in Section 6.1. This stage inserts a `BUBBLE` if the forwarded data is invalid and needs to stall the previous stage. The errors allowed to be emitted by this stage are `ECALL`, `EBREAK`[5], and `ILLEGAL_INSTRUCTION`.

5 `ECALL` and `EBREAK` are valid, but handled like errors. Therefore, the status needs to be set accordingly.

5.2.1 Instruction Decoder

The task of the Instruction Decoder is to interpret the instruction word retrieved from memory and provide control signals applicable to subsequent stages. All supported instructions are listed in Appendix A. Essential tasks include

- determining the operation to be executed,
- decoding the source register(s),
- identifying the destination register and
- parsing the immediate value[6].

6 The RISC-V ISA[1] specifies all instructions and how they are encoded.

The struct for a decoded instruction is depicted in Listing 5.1.

```
typedef struct packed {
    op::t op;
    logic [4:0] rd_address;
    logic [4:0] rs1_address;
    logic [4:0] rs2_address;
    csr::t csr;
    logic [31:0] immediate;
} t;
```

Listing 5.1: Instruction stuct.

The individual fields contain the following information:

`op` is the operation[7] to be performed.

`rd_address` holds the destination register address[8].

`rs1_address` holds the address of the first source register.

`rs2_address` holds the address of the second source register.

`csr` contains the CSR address.

`immediate` is the immediate data.

7 For the implementation, it is very helpful to use the *casez* statement (see Listing B.6).

8 If the source or destination register address is not specified in the instruction word, the address must be set to 0 to simplify forwarding.

Any unsupported instruction must output an `op::ILLEGAL`. This includes accessing unimplemented CSRs or writing to read-only CSRs (see RISC-V privileged spec [2] – Chapter 2.1). All implemented CSRs are listed in `defines/csr.sv`.

5.2.2 *Register File*

The Register File contains the CPU's 32 machine registers x0 – x31 (each 32 bit wide). It has one **synchronous** write port and two **asynchronous** read ports. The register at address zero **always** reads zero. The read and write ports are defined as follows:

`read_address1` holds the register address for the first read port.

`read_data1` contains the data from the register specified by `read_address1`.

`read_address2` holds the register address for the second read port.

`read_data2` contains the data from the register specified by `read_address2`.

`write_address` holds the register address for the write port.

`write_data` contains the data that must be stored in the register specified by `write_address`.

`write_enable` is the enable/disable signal for writing to registers.

5.3 Execute Stage

The Execute Stage performs all arithmetic operations. This includes general arithmetic op-codes but also comparisons and address calculations for branches, jumps, and memory access. This stage has the following inputs:

`instruction_in` is the decoded instruction from the Decode Stage.

`program_counter_in` is the memory address that corresponds to the current instruction.

`rs1_data_in` holds the data of the first operand to the instruction, if applicable.

`rs2_data_in` holds the data of the second operand of the instruction, if applicable.

Taking these input values, the stage generates the following outputs (all registered):

`instruction_out` provides the unmodified `instruction_in`.

`program_counter_out` provides the unmodified `program_counter_in`.

`next_pc_out` is the PC *after* the current instruction. This is the same as `jump_address_out` for unconditional jumps and taken branches, and `program_counter_in + 4` otherwise.

`rd_data_out` contains the data to write back to the `rd` register if it is already known after this stage. For load and store operations, this signal holds the memory address.

`source_data_out` holds the source data for store and CSR operations, if applicable.

Additionally, the Execute Stage must respect and generate the pipeline control signals as described in Section 6.1. The only allowed error to be emitted by this stage is `FETCH_MISALIGNED`[9]. It must also perform forwarding according to Section 6.2 for all already completed instructions.

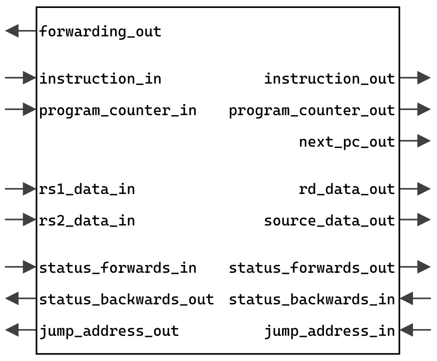

Figure 5.4: The pin-out of the Execute Stage.

9 The RISC-V ISA [1] specifies that instruction address misaligned exceptions occur on jumps, rather than on an instruction fetch.

5.4 *Memory Stage*

The Memory Stage performs all load and store operations. It mostly deals with controlling the wishbone interface, sign extension, and byte selection. This stage has the following inputs:

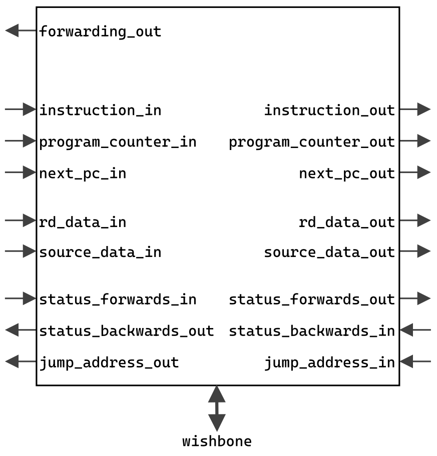

Figure 5.5: The pin-out of the Memory Stage.

`instruction_in` is the decoded instruction.

`program_counter_in` is the memory address corresponding to the current instruction.

`next_pc_in` is the address of the next instruction.

`rd_data_in` holds the data to write back to the `rd` register, if already known before this stage. For load and store operations, this signal holds the memory address.

`source_data_in` holds the data for store and CSR operations, if applicable.

Taking these input values, the stage generates the following outputs (all registered):

`instruction_out` provides the unmodified `instruction_in`.

`program_counter_out` provides the unmodified `program_counter_in`.

`next_pc_out` provides the unmodified `next_pc_in`.

`rd_data_out` contains the data to write back to the `rd` register, if it is already known after this stage.

`source_data_out` holds the source data for CSR operations, if applicable.

This stage can interact with the memory bus via that `wishbone`[10] interface. Both reads and writes may have side effects, so they cannot be aborted safely.

10 For a detailed description of the wishbone signals, refer to Section 6.3.

Additionally, the Memory Stage must respect and generate the pipeline control signals as described in Section 6.1. This stage will stall until the wishbone transaction is complete. The only allowed errors to be emitted by this stage are `LOAD_MISALIGNED`, `STORE_MISALIGNED`, `LOAD_FAULT`, and `STORE_FAULT`. It must also perform forwarding according to Section 6.2 for all already completed instructions.

5.5 Writeback Stage

The Writeback Stage performs all remaining operations not handled in earlier stages[11]. Additionally, it maintains the values of the Control and Status Registers (CSRs) and handles exceptions and interrupts. This stage has the following inputs:

`instruction_in` is the decoded instruction.

`program_counter_in` is the memory address corresponding to the current instruction.

`next_pc_in` is the address of the next instruction.

`rd_data_in` holds the data to write back to the `rd` register if it is already known before this stage.

`source_data_in` holds the data for CSR operations, if applicable.

`interrupts` contain the signals for external and timer interrupts, to be passed directly to the CSRs (see Section 5.5.1). The signals must remain high until the interrupts are handled in software.

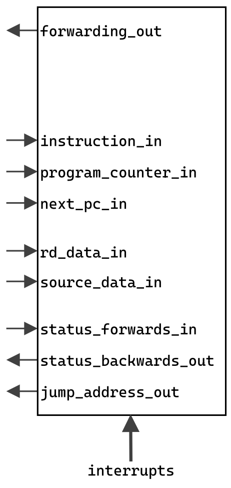

Figure 5.6: The pin-out of the Writeback Stage.

11 `WFI` and `FENCE` are implemented as `NOP`s; `ECALL` and `EBREAK` are implemented as exceptions; `FENCE.I` is implemented in the Writeback Stage as a jump to the next instruction.

This module does not generate any outputs besides what is needed to respect and generate the pipeline control signals as described in Section 6.1. This stage must never stall to ensure no ongoing memory operations are interrupted. The Writeback Stage will generate a jump in any of the following conditions:

- When a trap is taken due to an interrupt or exception, jump to `mtvec` (cf. Section 5.5.2).
- When `instruction_in` is an `MRET` instruction, jump to `mepc`.
- When `instruction_in` is a `FENCE.I` instruction, jump to `next_pc_in`.

It must also perform forwarding according to Section 6.2 for all instructions.

5.5.1 Control and Status Registers

The CSRs are used to control and observe the machine state. The following registers[12] must be implemented:

12 This is the minimal set of registers required for a machine-mode-only processor [2].

`MSTATUS` represents the core operating state. The `MPIE` and `MIE` bits must be readable and writable; all other bits must read 0. `MIE` must be cleared on reset.

`MTVEC` holds the jump address for interrupts and exceptions (traps). It can be read or written with arbitrary values, but the lowest two bits must be 0 (to keep the address aligned).

`MIP` signals pending interrupts. Only `MEIP` and `MTIP` are implemented and are read-only.

`MIE` enables and disables specific interrupts. Only `MEIE` and `MTIE` are implemented and readable and writable.

MCYCLE/MCYCLEH represents a 64-bit counter, with the 32 most significant bits in MCYCLEH and the others in MCYCLE. Both registers are readable and writable and support arbitrary values. If not currently written, the counter increments by one every clock cycle.

MINSTRET/MINSTRETH represents a 64-bit counter, with the 32 most significant bits in MINSTRETH and the others in MINSTRET. Both registers are readable and writable and support arbitrary values. If not currently written, the counter increments by one whenever a valid instruction completes.

MSCRATCH is a scratch register with no special behavior. It is readable and writable, and supports any value.

MEPC stores the address of the instruction that caused an exception. It can be read or written with arbitrary values, but the lowest two bits must be 0 (to keep the address aligned).

MCAUSE stores the reason a trap was taken. It is readable and writable, and supports any value. On reset, this register is to be cleared.

All other CSRs[13] are read-only zero.

13 **Hint**: Creating a dedicated submodule for handling all the CSRs might be helpful.

5.5.2 *Interrupt Handling*

MEIP is set while external_interrupt is asserted. MTIP is set while timer_interrupt is asserted. Every exception causes an immediate trap once it reaches the Writeback Stage. An interrupt causes a trap if one of the following[14] is true:

- MEIP, MEIE, and MIE are set when an instruction completes[15].
- MTIP, MTIE and MIE are set when an instruction completes.

14 If any of the flags are changed by the current instruction, the updated values must be used.

15 "Completes" means with a VALID or ERROR status (**no** BUBBLE).

If a trap occurs, a couple of things must happen at once:

- The Writeback Stage triggers a jump to the address stored in MTVEC.
- MCAUSE is updated according to Section 3.1.15 in [2].
- MEPC is updated with the current PC (for exceptions) or the next PC (for interrupts).
- MPIE is set to MIE.
- MIE is cleared.

Once the trap handler completes, it usually returns to normal instruction flow by executing an MRET instruction. When an MRET instruction occurs, the following changes[16] happen:

- The Writeback Stage triggers a jump to the address stored in MEPC.
- MIE is set to MPIE.
- MPIE is set.

16 Beware that if an interrupt is pending, it may trigger immediately after an MRET but still in the same clock cycle!

6 Additional Information

6.1 Pipeline Status

In the HADES-V pipeline control system, we utilize two types of status indicators to monitor the progress of instructions through the pipeline[1]: a Forwards Status that indicates a stage's progress to the next stage, and a Backwards Status that indicates the current stage's readiness for input from the stage before[2].

1 The passing of the status signals (and the jump address) can be seen in the lower 3 signals of every stage in Figure 5.1.

2 **Hint**: Consider sequential logic for the Forwards Status unlike the Backwards Status.

6.1.1 Forwards Status

This status signal is passed from an earlier to a later pipeline stage, e.g., from the Decode Stage to the Execute Stage. There are 3 different types of Forwards Status:

`VALID` means all output signals of the current stage are valid. The following stage can process the signals of the current stage normally.

`BUBBLE` means the output signals are invalid. The following stage must ignore all other signals it gets from the current stage.

All other values in this status mean the output signals (other than the PC) are invalid, and an exception occurred. The following stage passes this state and the PC to the next stage and ignores all other signals.

The possible values for the Forwards Status are defined in Listing 6.1.

```
typedef enum [3:0] {
    VALID,
    BUBBLE,
    FETCH_MISALIGNED,
    FETCH_FAULT,
    ILLEGAL_INSTRUCTION,
    LOAD_MISALIGNED,
    LOAD_FAULT,
    STORE_MISALIGNED,
    STORE_FAULT,
    ECALL,
    EBREAK
} forwards_t;
```

Listing 6.1: Forwards Status enum.

Each stage only processes an input when the forwards status is `VALID`[3], except the Writeback Stage, which needs to handle the `ERROR` state.

3 Pay close attention to forwarding signals, such as `address` and `data_valid`, since they make only sense if the input state is `VALID`.

6.1.2 Backwards Status

Contrary to the status signal explained before, the Backwards Status signal is passed from a later pipeline stage back to an earlier one, e.g., from the Execute Stage to the Decode Stage. There exist 3 different types of Backwards Status:

`READY` means the current stage has finished processing its inputs and is ready to accept new ones. Hence, the stage before may provide new output signals.

`STALL` means the current stage is still busy processing its inputs and cannot accept new ones. Hence, the stage before must maintain its current output values, unless it indicates a `BUBBLE`.

`JUMP` means the current stage (or a later stage) indicates a jump to `jump_address_backwards`. Hence, the stage before must abort its current instruction and indicate a `BUBBLE`. The Instruction Fetch Stage must continue to fetch instructions from `jump_address_backwards` after the `BUBBLE`.

The possible values for the Backwards Status are defined in Listing 6.2.

```
    typedef enum [1:0] {
        READY,
        STALL,
        JUMP
    } backwards_t;
```

Listing 6.2: Backwards Status Enum.

This status signal must be handled *before* changing any registers; therefore, it must be passed through all stages[4] without any delay. Thus, it exclusively consists of combinatorial logic. This rule also applies to `jump_address_backwards`. This is crucial since the Instruction Fetch Stage is responsible for setting the PC to this address on the subsequent positive clock edge if a `JUMP` is requested.

4 The Backwards Status from a later pipeline stage takes precedence over a status from an earlier one; e.g., if Memory Stage stalls and the Execute Stage wants to jump, the Execute Stage must forward the `STALL` rather than its own `JUMP`.

6.2 Forwarding

To mitigate penalties arising from data dependencies in the pipeline, we use Forwarding. Within this method, the outcomes of each stage are directed to a Forwarding Unit positioned within the Decode Stage (see Section 5.2). The forwarding of data involves a signal[5] specific to each stage (Execute, Memory, and Writeback), which is also the input for the output register of that particular stage.

Since results might not be immediately available, the *forwarding* signal is a struct containing the following signals:

`address` denotes the destination register address where the `data` is to be stored. If `address` is zero, the other signals should be ignored.

`data` holds the result, i.e., the data to be stored in the corresponding register.

5 Rather than directly retrieving results from the output register of each stage, this approach aims to reduce penalties incurred due to invalid data while simplifying the functioning of the Forwarding Unit.

`data_valid` indicates the validity of `data`.

The struct of the forwarding signal is shown in Listing 6.3.

```
    typedef struct packed {
        logic        data_valid;
        logic [31:0] data;
        logic  [4:0] address;
    } t;
```

Listing 6.3: Forwarding struct.

The Forwarding Unit must check if the current instruction relies on data from instructions in the Execute, Memory, or Writeback Stage[6]. If this is the case, instead of taking the data from the Register File, it must use the *forwarded* result that is still in the pipeline. The Forwarding Unit is also responsible for checking the `data_valid` signal when forwarding the result[7].

6 If the source register of this instruction equals the destination register of instructions in the pipeline.

7 The register at address zero (x0) can always be forwarded, as it is always zero.

6.3 Wishbone

Wishbone is an open interconnect standard by OpenCores [9]. It is used to connect the CPU core to its peripherals, and in the HADES-V architecture also the RAM. This section briefly explains the operation of this bus.

The HADES-V architecture uses classic wishbone bus cycles with support for synchronous and asynchronous slaves.

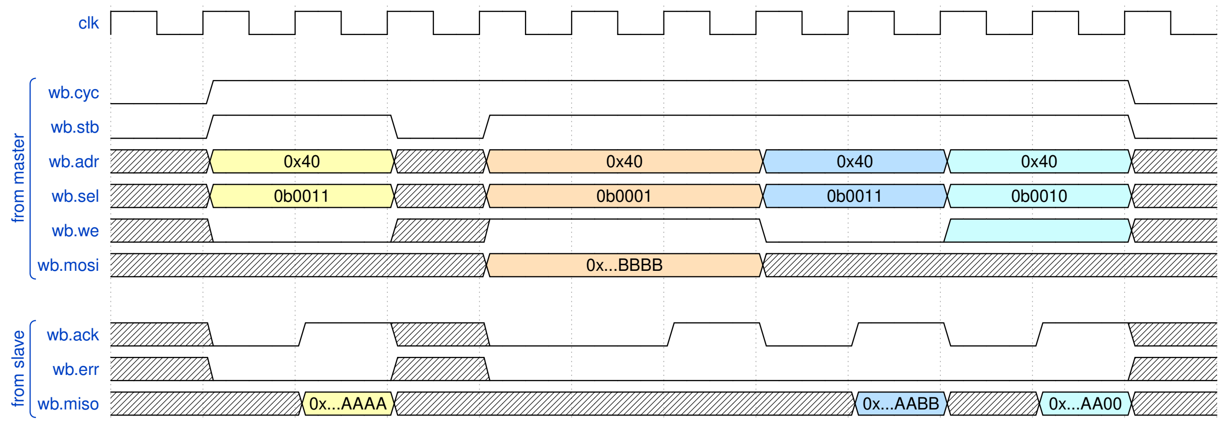

Figure 6.1: A wishbone cycle consisting of 4 separate transfers with a synchronous slave.

Figure 6.1 shows an example wishbone bus cycle consisting of 4 transfers. All signals are read at the rising clock edge. The bus master must hold the `cyc` signal high until the last transfer in the cycle is complete.

The bus master starts a transfer by setting `stb` (possibly in the same cycle as `cyc`. Simultaneously, all other signals driven by the master must be valid. The `adr` signal indicates the *word address* (not the byte address) that is accessed. The bits in the `sel` signal indicate which bytes of the word are accessed. The `we` signal indicates a write transfer if set, or a read transfer otherwise. Finally, the `dat_mosi` signal carries the data for write transfers.

The slave corresponding to the specified address must then respond to this transfer. The `ack` and `err` signals must be low until the transfer is complete. Then either of them must be asserted by the slave for one clock cycle to indicate a success (`ack`) or an error (`err`). On a successful read transfer, `dat_miso` carries the data while `ack` is high.

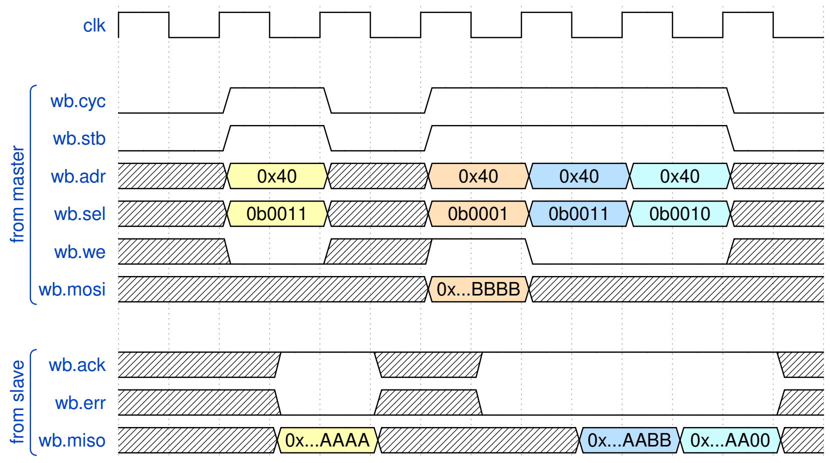

Figure 6.2: The same wishbone cycle with a fast, asynchronous slave.

To increase performance, slaves may generate their acknowledge signal asynchronously. Figure 6.2 shows the same wishbone cycle as before, except that the slave responds asynchronously. This allows to complete a memory transfer in a single cycle and is used for them main memory in the HADES-V architecture.

6.4 Development Board

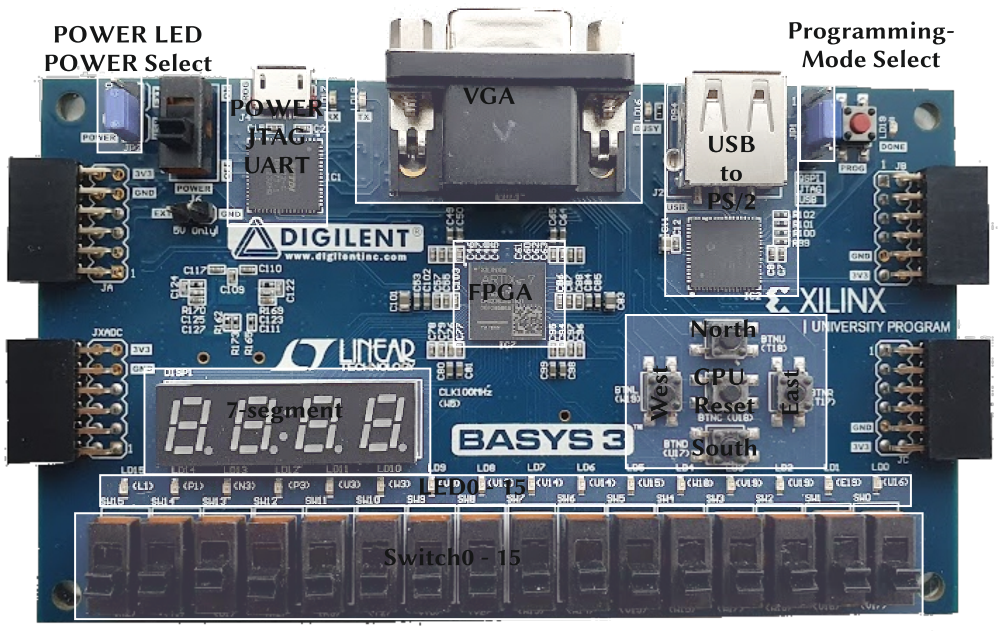

Figure 6.3: The Basys3 board.

The Basys3 board [3] is a complete, ready-to-use digital circuit development platform based on an Artix®-7 FPGA from AMD. The board offers the following ports and peripherals:

- 16 user switches

- 16 user LED
- 5 user push buttons
- 4-digit 7-segment display
- 3 Pmod connectors
- Pmod for ADC signals
- 12-bit VGA output
- USB-UART bridge
- serial flash
- Digilent USB-JTAG port
- USB HID (USB to PS/2)

Power Supplies The board is powered by the Digilent USB-JTAG port (J4) or an external 5 V power supply. Jumper JP3 determines, which source is used. A power LED (LD20), driven by the "power good" output of the LTC3633 supply, indicates that the supplies are turned on and operating normally.
The USB port delivers enough power for the vast majority of designs. A few demanding applications, including any that drive multiple peripheral boards, might require more power than the USB port can provide. An external power supply can be used by using the external power header (J6) and setting jumper JP2 to "EXT". Power supplies must offer voltage ranging from 4.5 V_{DC} to 5.5 V_{DC} and at least 1 A of current (i.e., at least 5 W of power).

FPGA Configuration After power-on, the Artix®-7 FPGA must be configured before it can perform any functions. The on-board jumper (JP1) selects between three different configurations modes:

1. a computer can use the Digilent USB-JTAG port (J4, labeled "PROG") to configure the FPGA any time the power is on
2. a file stored in the non-volatile serial flash device can be transferred to the FPGA by using the SPI port
3. a configuration file can be transferred from a USB memory stick attached to the USB HID port

Oscillators / Clocks Basys3 has a single 100 MHz oscillator connected to pin W5 (W5 is a MRCC input on bank 34). The input clock can drive MMCM or PLL to generate clocks of various frequencies and with known phase relationships that may be needed throughout a design.

Basic I/O (LED, Switches, Buttons) Figure 6.4 shows the basic I/O devices of the Basys3 board. It includes sixteen individual LED, which are anode-connected to the FPGA via 330 Ω resistors and can be switched on with high signals.
The sixteen switches are located at the bottom of the board. They (and the five buttons above) are connected to the FPGA via serial resistors to prevent damage if accidentally defined as output.

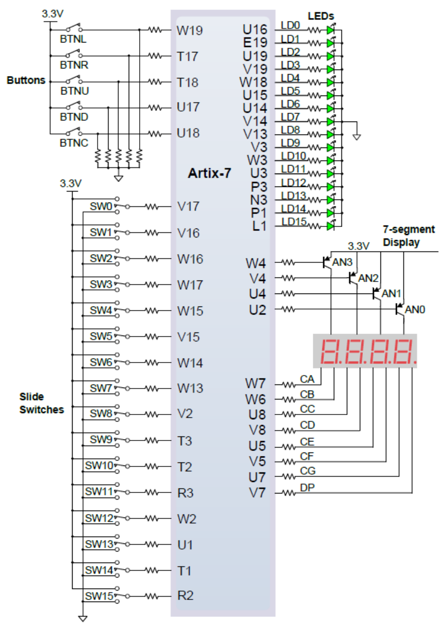
3.3V
Buttons
BTNL
BTNR
BTNU
BTND
BTNC
W19
T17
T18
U17
U18
Artix-7
LEDs
U16 E19 U19 V19 W18 U15 U14 V14 V13 V3 W3 U3 P3 N3 P1 L1
LD0 LD1 LD2 LD3 LD4 LD5 LD6 LD7 LD8 LD9 LD10 LD11 LD12 LD13 LD14 LD15
Slide Switches
SW0 SW1 SW2 SW3 SW4 SW5 SW6 SW7 SW8 SW9 SW10 SW11 SW12 SW13 SW14 SW15
V17 V16 W16 W17 W15 V15 W14 W13 V2 T3 T2 R3 W2 U1 T1 R2
7-segment Display
W4 V4 U4 U2
AN3 AN2 AN1 AN0
W7 W6 U8 V8 U5 V5 U7 V7
CA CB CC CD CE CF CG DP

Figure 6.4: Basic I/O devices (from [3]).

Seven-Segment Display The Basys3 board provides a four-digit common anode seven-segment LED display. Every digit uses one common anode for every digit. In total, the circuit has four anodes and 7 cathodes for the whole display. The cathodes are shared between all four digits, which results in a multiplexed display design. The update rate between the single digits must be higher than 45 Hz to prevent a flickering display. All four digits should be driven once every 1 to 16 ms (1 kHz to 60 Hz)

USB HID Host The auxiliary function microcontroller (Microchip PIC24FJ128) provides the Basys3 board with USB HID host capability. After power-up, the microcontroller is in configuration mode, either downloading a bitstream to the FPGA or waiting for it to be configured from other sources. Once the FPGA is configured, the microcontroller switches to application mode, which in this case is the USB HID host mode. The firmware in the microcontroller can drive a mouse or a keyboard attached to the type A USB connector at J2 labeled "USB".

VGA Beside the LED and the seven-segment display, another visual output is available. The VGA port with 4 bits per color supports up to 4096 different colors. The provided VGA peripheral for the Wishbone bus uses only one bit per color and one intensity bit (necessary reduction to save valuable memory). Additional to the color information are two more signals required. One signal for vertical synchronization and one for the horizontal synchronization. The Basys3 board board uses 14 FPGA pins to create the VGA port as shown in Figure 6.5.

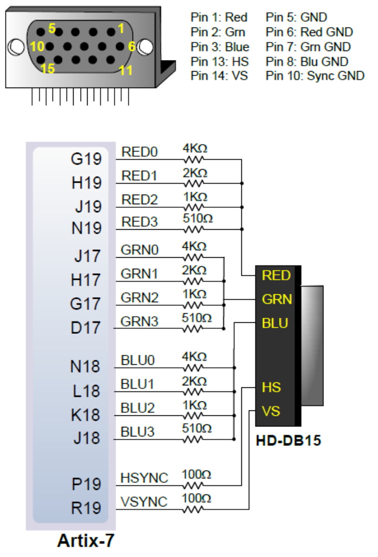

Figure 6.5: Pin configuration for the VGA port (from [3]).

Table 6.1 shows the connected FPGA pins with the SystemVerilog top level ports.

Name	FPGA pin	Type	Connected with	Description
clk	W5	in	clk_100mhz	onboard clock (100 MHz)
btn_in<0>	U18	in	buttons_async(0)	button center
btn_in<1>	T18	in	buttons_async(1)	button north
btn_in<2>	W19	in	buttons_async(2)	button west
btn_in<3>	T17	in	buttons_async(3)	button east
btn_in<3>	U17	in	buttons_async(3)	button south
led_out<0>	U16	out	leds(0)	LED output 0
led_out<1>	E19	out	leds(1)	LED output 1
led_out<2>	U19	out	leds(2)	LED output 2
led_out<3>	V19	out	leds(3)	LED output 3
led_out<4>	W18	out	leds(4)	LED output 4
led_out<5>	U15	out	leds(5)	LED output 5
led_out<6>	U14	out	leds(6)	LED output 6
led_out<7>	V14	out	leds(7)	LED output 7
led_out<8>	V13	out	leds(8)	LED output 8
led_out<9>	V3	out	leds(9)	LED output 9
led_out<10>	W3	out	leds(10)	LED output 10
led_out<11>	U3	out	leds(11)	LED output 11
led_out<12>	P3	out	leds(12)	LED output 12
led_out<13>	N3	out	leds(13)	LED output 13
led_out<14>	P1	out	leds(14)	LED output 14
led_out<15>	L1	out	leds(15)	LED output 15
swt_in<0>	V17	in	switches_async(0)	switch 0
swt_in<1>	V16	in	switches_async(1)	switch 1
swt_in<2>	W16	in	switches_async(2)	switch 2
swt_in<3>	W17	in	switches_async(3)	switch 3
swt_in<4>	W15	in	switches_async(4)	switch 4
swt_in<5>	V15	in	switches_async(5)	switch 5
swt_in<6>	W14	in	switches_async(6)	switch 6
swt_in<7>	W13	in	switches_async(7)	switch 7
swt_in<8>	V2	in	switches_async(8)	switch 8
swt_in<9>	T3	in	switches_async(9)	switch 9
swt_in<10>	T2	in	switches_async(10)	switch 10
swt_in<11>	R3	in	switches_async(11)	switch 11
swt_in<12>	W2	in	switches_async(12)	switch 12
swt_in<13>	U1	in	switches_async(13)	switch 13
swt_in<14>	T1	in	switches_async(14)	switch 14
swt_in<15>	R2	in	switches_async(15)	switch 15
uart_rx	B18	in	uart_rx_async	RS232 RXD
uart_tx	A18	out	uart_tx	RS232 TXD
vga_r<0>	G19	out	vga_red(0)	VGA red bit 0
vga_r<1>	H19	out	vga_red(1)	VGA red bit 1
vga_r<2>	J19	out	vga_red(2)	VGA red bit 2
vga_r<3>	N19	out	vga_red(3)	VGA red bit 3
vga_b<0>	N18	out	vga_blue(0)	VGA blue bit 0
vga_b<1>	L18	out	vga_blue(1)	VGA blue bit 1
vga_b<2>	K18	out	vga_blue(2)	VGA blue bit 2
vga_b<3>	J18	out	vga_blue(3)	VGA blue bit 3
vga_g<0>	J17	out	vga_green(0)	VGA green bit 0
vga_g<1>	H17	out	vga_green(1)	VGA green bit 1
vga_g<2>	G17	out	vga_green(2)	VGA green bit 2
vga_g<3>	D17	out	vga_green(3)	VGA green bit 3
vga_hsync	P19	out	vga_hsync	VGA horizontal sync
vga_vsync	R19	out	vga_vsync	VGA vertical sync
seg<0>	W7	out	segments(0)	7 Segment bit a
seg<1>	W6	out	segments(1)	7 Segment bit b
seg<2>	U8	out	segments(2)	7 Segment bit c
seg<3>	V8	out	segments(3)	7 Segment bit d
seg<4>	U5	out	segments(4)	7 Segment bit e
seg<5>	V5	out	segments(5)	7 Segment bit f
seg<6>	U7	out	segments(6)	7 Segment bit g
dp	V7	out	segments(7)	7 Segment decimal point
an<0>	U2	out	segments_select(0)	7 Segment anode 1
an<1>	U4	out	segments_select(1)	7 Segment anode 2
an<2>	V4	out	segments_select(2)	7 Segment anode 3
an<3>	W4	out	segments_select(3)	7 Segment anode 4

Table 6.1: All the pins of the FPGA and their connections.

6.5 Peripherals

The wishbone (see Section 6.3) is connected with the data memory and many peripherals. This section introduces all offered HADES-V peripherals. In Table 6.2, all peripherals and the corresponding address spaces are listed.

Module	Byte address	Bit	Description
Memory (RAM)	0x10000...0x12000		Read/Write to RAM
LEDs	0x80000	16...0	Set individual bit to turn on/off the corresponding LED
Buttons	0x81000	4	Status of *center* button (pressed = '1')
	0x81000	3	Status of *north* button (pressed = '1')
	0x81000	2	Status of *west* button (pressed = '1')
	0x81000	1	Status of *east* button (pressed = '1')
	0x81000	0	Status of *south* button (pressed = '1')
Switches	0x82000	16...0	Status of the corresponding switch (switch enabled = '1')
Segments	0x83000	31...24	7-Segments of 1000
(see Section 6.5.1)	0x83000	23...16	7-Segments of 100
	0x83000	15...8	7-Segments of 10
	0x83000	7...0	7-Segments of 1
UART	0x84000	31...24	Transmitt Status
(see Section 6.5.2)	0x84000	23...16	Receive Status
	0x84000	15...8	unused
	0x84000	7...0	RX/TX - Buffer
Timer	0x85000		MTIME Status
(see Section 6.5.3)	0x85001		MTIME
	0x85002		MTIMEH
	0x85003		MTIMECMP
	0x85004		MTIMECMPH
VGA	0x90000...0x99600		VGA-Buffer ($640 \cdot 480 \cdot 4$ bit)
(see Section 6.5.4)			
Test	0x120000		Test register (write to send a message to testbench)
(see Section 6.5.5)	0x120001		Interrupt register (down counter)
	0x120002		Counter register (reads return an incrementing number)
	0x120003		Stall Acknowledge register (delayed acknowledge)
	0x120004		Stall Error register (delayed error)

Table 6.2: All the peripherals and their corresponding addresses.

6.5.1 Segments

To control the individual segments and dots[8] on the four seven-segment displays, the designated 32-bit memory location is used. Each display is managed by a specific set of 8 bits within this address: the rightmost display uses the lowest 8 bits, the next 8 bits control the tens place, the following 8 bits handle the hundreds place, and the top 8 bits manage the thousands place. Each bit corresponds to a segment:

Bit index	7	6	5	4	3	2	1	0
Segment	Dot	G	F	E	D	C	B	A

8 The colon between the first two and the second two digits is not connected in hardware. Therefore, it cannot be controlled by software.

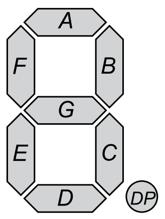

Figure 6.6: 7 Segment display.

6.5.2 UART

The UART peripheral is a serial communication interface (RS232) for communicating with a working station. The baud rate is fixed set to 115 200 kbit/s.

The transmit and receive buffer is 1 Byte large and is represented in big-endian format: the most significant byte will be sent first. Upon reading from the buffer, it retrieves the most recent received byte, while writing to the buffer stores the subsequent byte for transmission. Transmission occurs automatically when the transmission unit is idle and a new byte is available for sending.

Alongside the UART buffer, there exist control and status bits[9] that manage the sending and receiving processes:

- TX buffer empty: This bit is set if the buffer is empty and cleared upon writing to the buffer.
- TX interrupt enable: Set this bit to trigger an interrupt when the TX buffer is empty.
- TX error: This bit is set when a buffer overflow occurs (overwriting the full TX buffer) and cleared upon read.
- RX buffer full: This bit is set if the buffer is full and cleared upon reading from the buffer.
- RX interrupt enable: Set this bit to trigger an interrupt when the RX buffer is full.
- RX error: This bit is set if a buffer overflow occurs (full RX buffer gets overwritten due to new received byte) and cleared upon read.

9 Since RISC-V supports byte (8-bit), halfword (16-bit), and word (32-bit) memory operations, it allows reading or writing only particular parts of a memory address.

	TX Status				**RX Status**				-	**Buffer**
Bit Index	31:27	26	25	24	23:19	18	17	16	15:8	7:0
Functionality	xxxxx	EMPTY	IE	ERR	xxxxx	FULL	IE	ERR	xxxxxxxx	BUFFER

6.5.3 *Timer*

The timer module is responsible for the two memory mapped timer registers `mtime` and `mtimecmp` defined in the privileged RISC-V ISA [2] (3.2.1 Machine Timer Registers). In order to provide a mechanism for determining the period of one tick, an additional read-only `"MTIME Status"` register has been implemented, wherein the lowest 8 bits denote the nanoseconds per system clock cycle.

6.5.4 *VGA*

The VGA peripheral controls the graphical output on the VGA output. The output resolution is predefined to 640*x*480 pixel with 16 colors. The used color system is the Color Graphics Adapter or iRGB system (`https://en.wikipedia.org/wiki/Color_Graphics_Adapter`). It is a 4-bit color system. Beside the 3 primary colors, red, green and blue, an intensity bit exists describing the intensity of the RGB color. The next table shows the assignment of the individual bits:

Bit	**Descpription**
0	blue
1	green
2	red
3	intensity

With 4 bits it is possible to generate 16 different colors. In the next table, all possible colors are shown. The table shows that the intensity bit distinguishes the intensity of the RGB color. The only exceptions are the brown and yellow color. The standard defines to generate a brown color instead of a dark yellow, for a better distinction of the two colors.

Color	Bits	Color	Bits	Color	Bits	Color	Bits
black	0000	red	0100	gray	1000	light red	1100
blue	0001	magenta	0101	light blue	1001	light magenta	1101
green	0010	brown	0110	light green	1010	yellow	1110
cyan	0011	light gray	0111	light cyan	1011	white	1111

Since the screen has $640\,\text{pixel} \cdot 480\,\text{pixel} = 307200\,\text{pixel}$ and each pixel needs 4 bit, the graphics memory has a size of 153 600 B. The first graphics memory address is the pixel on the top left $(x = 0,\ y = 0)$ of the screen. The rising address goes from the left to the right til the end of the row. Then it jumps to the next row and starts on the left. Therefore, the next pixel of address $n \cdot (640 - 1), n \in \{0, \ldots, 478\}$ $(x = 639,\ y = n)$ is on the next row on the left $(x = 0,\ y = n + 1)$. In a single memory location, the LSBs correspond to the lowest pixel index, while the MSBs correspond to the highest pixel index within that specific memory location[10]:

10 Since RISC-V supports byte (8-bit), halfword (16-bit), and word (32-bit) memory operations, it allows reading or writing only particular parts of a memory address.

	31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
Address + 0	px 7	px 6	px 5	px 4	px 3	px 2	px 1	px 0
Address + 1	px 15	px 14	px 13	px 12	px 11	px 10	px 9	px 8
...	...							

6.5.5 Test

The test peripheral is designed for CPU testing and comprises the following registers:

- **Test Register**: Writing to this register sends a message to the testbench.
  - `0x0000`: Indicates a passed test case.
  - `0x0001`: Indicates a failed test case.
  - `0x0002`: Halts the simulation.
- **Interrupt Register**: Contains a down counter that triggers an interrupt when it reaches 0.
- **Counter Register**: Reading from this register returns an incrementing number, starting from 0.
- **Stall Acknowledge Register**: Read and write operations to this register waits for 3 clock cycles before acknowledging.
- **Stall Error Register**: Read and write operations to this register waits for 3 clock cycles before raising a wishbone error.

A *Instruction Set Listing*

The following table lists all instructions that are implemented by the HADES-V CPU. This is a subset of the RISC-V ISA [1]. All instructions not in this table raise an “Illegal Instruction” exception.

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt[4:0]	rs1	001	rd	0010011	SLLI
0000000	shamt[4:0]	rs1	101	rd	0010011	SRLI
0100000	shamt[4:0]	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
imm[11:0]		rs1	000	rd	0001111	FENCE
imm[11:0]		rs1	001	rd	0001111	FENCE.I
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK
001100000010		00000	000	00000	1110011	MRET
000100000101		00000	000	00000	1110011	WFI
csr		rs1	001	rd	1110011	CSRRW
csr		rs1	010	rd	1110011	CSRRS
csr		rs1	011	rd	1110011	CSRRC
csr		uimm[4:0]	101	rd	1110011	CSRRWI
csr		uimm[4:0]	110	rd	1110011	CSRRSI
csr		uimm[4:0]	111	rd	1110011	CSRRCI

B Coding Style

In the following sections we provide some short and basic examples on how to implement certain structures, and it is highly recommended to follow this coding style, as code quality is part of your assessment.

Important Note

For coding, use *combinatorial logic*, `always_comb` and `always_ff` blocks exclusively, while ensuring all time-dependent signals are only sensitive to the *positive clock edge*. In SystemVerilog, we exclusively utilize the `logic` datatype (never `reg` or `wire`).

B.1 Signal Declaration and Assignment

When assigning a value to a signal, it is crucial to employ the assign statement to ensure the value is assigned combinatorially rather than set only at startup, as illustrated in Listing B.1.

```
// This line assigns the value only once during startup
logic my_signal_wrong = SOMETHING;
// This line assigns the value combinatorially
logic my_signal;
assign my_signal = SOMETHING;
```

Listing B.1: Signal decleration and assignment.

B.2 Module Instantiation

It is highly recommended to connect **all** ports when instantiating a module, even if the ports have the same name. The following code snippets show a subcomponent (cf. Listing B.2) that is instantiated in a top module in Listing B.3.

```
module subcomponent (
    input  logic clk,
    input  logic rst,
    input  logic [31:0] data_in,
    output logic [31:0] data_out
);
[...]
endmodule
```

Listing B.2: Subcomponent module.

```
module top (
);

logic clk, rst;
logic [31:0] data_in;
logic [31:0] processed_data;

// Instantiate subcomponent module
subcomponent module_name (
    .clk(clk),
    .rst(rst),
    .data_in(data_in),
    .data_out(processed_data)
);

endmodule
```

Listing B.3: Instantiate a submodule.

B.3 Registers

Whenever implementing CPU registers, the recommended way doing so is shown in Listing B.4.

```
logic [31:0] test_reg;
always_ff @(posedge clk) begin
    if (rst) begin
        // Reset register
        test_reg <= RESET_VALUE;
    end
    else begin
        // Assign value to register
        if (condition)
            test_reg <= SOME_VALUE;
        else
            test_reg <= DEFAULT_VALUE;
    end
end
```

Listing B.4: A simple Register.

This design showcases registers with a synchronized reset (if needed) and updates values on the rising clock edge.Splitting the reset and value-setting actions into separate blocks (possibly using an "if-else" structure) is recommended. This separation enhances code clarity and simplifies understanding by clearly distinguishing between reset operations and value assignments, making the code easier to manage and maintain.

B.4 *Multiplexers*

When selecting among different input signals, a Multiplexer (MUX) is the hardware element of choice. To simplify the MUX implementation and improve clarity over multiple *if-else* conditions, it is recommended to use a *case* statement. The code snippet in Listing B.5 demonstrates a 4-to-1 MUX.

```
logic [31:0] mux_data_out;
logic  [2:0] mux_select;
always_comb begin
    case(mux_select)
        2'b00:   mux_data_out = input_data_0;
        2'b01:   mux_data_out = input_data_1;
        2'b10:   mux_data_out = input_data_2;
        2'b11:   mux_data_out = input_data_3;
        default: mux_data_out = 0; // Default case
    endcase
end
```

Listing B.5: A simple 4-1 MUX.

When the select signals need to deal with don't-care conditions the *casez*[1] statement (cf. Listing B.6) is helpful. Here, 'z' and '?' are considered don't-care values, and the expression always matches if the defined part matches. This is particularly useful for scenarios where certain bits are irrelevant or unspecified (e.g., within the Instruction Decoder in Section 5.2).

1 We do not recommend using `casex` since it is error prone (x-propagation).

```
logic [31:0] mux_data_out;
logic [31:0] mux_select;
always_comb begin
    casez(mux_select)
        {22'b?, 5'b00000, 5'b00000}: mux_data = data_0;
        {24'b?,   3'b101, 5'b00000}: mux_data = data_1;
        {27'b?,           5'b01010}: mux_data = data_2;
        {27'b?,           5'b11111}: mux_data = data_3;
        default:                     mux_data = 0;
    endcase
end
```

Listing B.6: A MUX with don't-care values.

List of Abbreviations

CPU	Central Processing Unit
CSR	Control and Status Register
FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
ISA	Instruction Set Architecture
MCU	Microcontroller Unit
MUX	Multiplexer
PC	Program Counter
RAM	Random-Access Memory
RISC	Reduced Instruction Set Computing
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver Transmitter

Bibliography

[1] RISC-V Foundation, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*, December 2019. [online] `https://github.com/riscv/riscv-isa-manual/releases/tag/Ratified-IMAFDQC`.

[2] RISC-V International, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203*, December 2021. [online] `https://github.com/riscv/riscv-isa-manual/releases/tag/Priv-v1.12`.

[3] Digilent, Inc., *Basys 3 Reference Manual.* [online] `https://reference.digilentinc.com/reference/programmable-logic/basys-3/reference-manual`.

[4] IEEE, "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language," *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, 2018. [online] `https://doi.org/10.1109/IEEESTD.2018.8299595`.

[5] D. Thomas, *Logic Design and Verification Using SystemVerilog (Revised).* CreateSpace Independent Publishing Platform, 2016. ISBN: 978-1523364022.

[6] Intel Cooperation, *Hexadecimal Object File Format Specification*, 1998. [online] `https://archive.org/details/IntelHEXStandard`.

[7] T. Scheipel, *Advances in Dynamic and Reconfigurable Embedded Systems Design.* PhD thesis, Graz University of Technology, Dec. 2022. [online] `https://www.scheipel.com/diss`.

[8] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, 6 ed., 2019. ISBN: 978-0-12-811905-1.

[9] OpenCores, *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, 2010. [online] `https://cdn.opencores.org/downloads/wbspec_b4.pdf`.