CS-200 Infrastructure Tutorial

Setting up the work platform

You are expected and encouraged to work on physical machines in INF 3 or remote desktops through virtual desktop infrastructure (VDI), both of which are equipped with all the necessary software.

Using physical machines in INF 3

1. Make your way to INF 3.
2. Log into any available desktop using your GASPAR credentials.

Using remote desktops through VDI

You can access the VDI service by either the VMware Horizon Client or a web browser. *Please note that the browser version is more limited than the VMware Horizon Client.*

VMware Horizon Client

1. Navigate to https://vdi.epfl.ch.
2. Click on the link `Install VMware Horizon Client` on the left to download the installer (this page may vary across operating systems).
3. Run the installer to install Omnissa Horizon Client on your machine.
4. Launch Omnissa Horizon Client (also named `VMware Horizon Client` in some platforms).
5. Enter `https://vdi.epfl.ch` as the connection server.
6. Log in with your GASPAR credentials.
7. Select and connect to the `IC-CO-IN-SC-INJ-2025-fall` or `IC-CO-IN-SC-MA-2025-fall` machine.

Web Browser

1. Navigate to https://vdi.epfl.ch.
2. Click on `VMware Horizon HTML Access`.

Figure 1: Starting page of https://vdi.epfl.ch

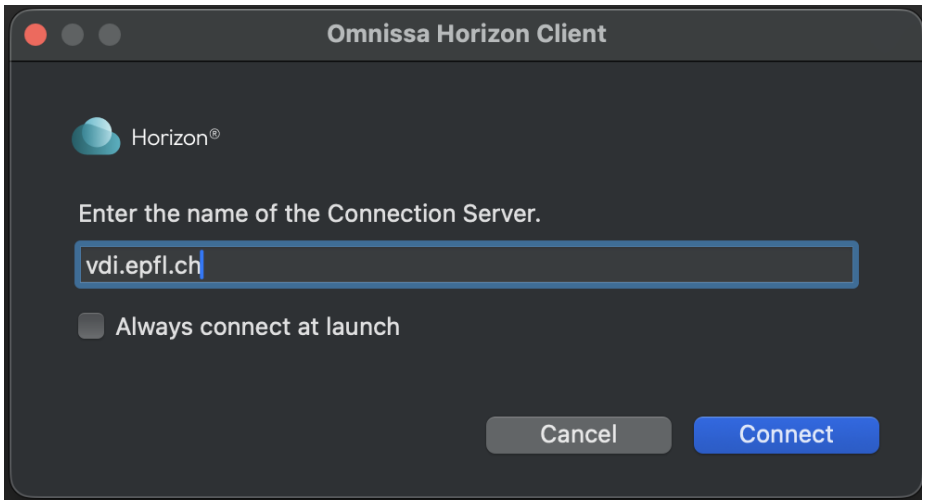

Figure 2: Entering the name of the connection server

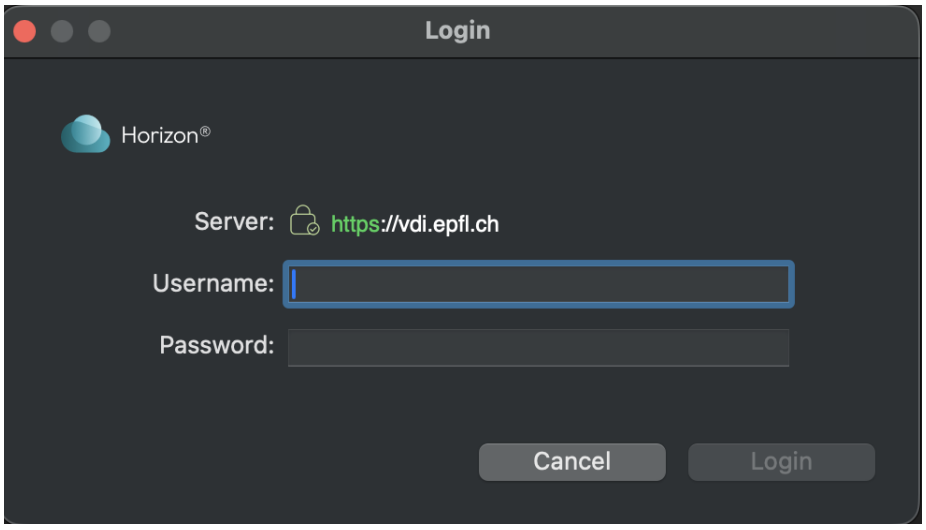

Figure 3: Logging in

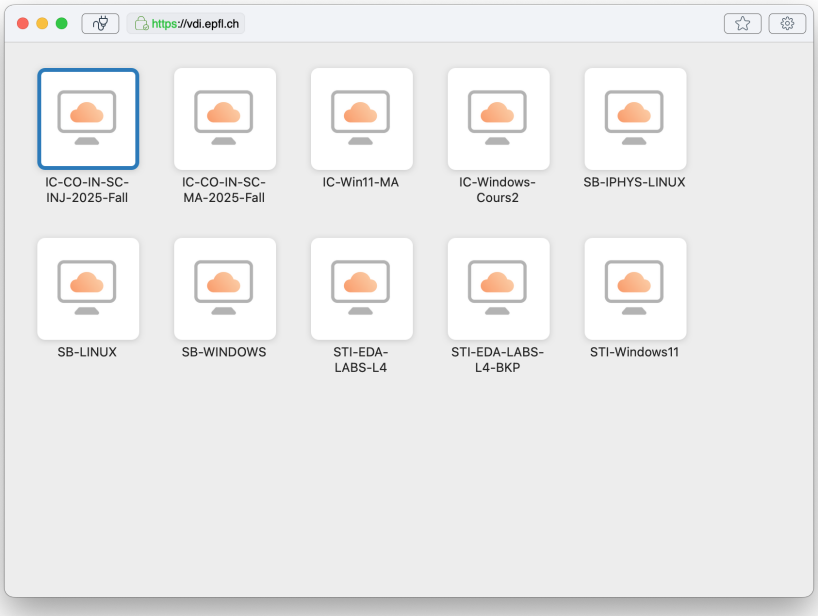

Figure 4: Main UI of the Omnissa Horizon Client

Figure 5: Starting page of https://vdi.epfl.ch

3. Log in with your GASPAR credentials.
4. Select and connect to the `IC-CO-IN-SC-INJ-2025-fall` or `IC-CO-IN-SC-MA-2025-fall` machine.

After you successfully log in to the machine, you will see a desktop similar to the following:

Under the hood, the physical machines in INF 3 are actually connecting to the same remote desktops as you would via the VDI. Therefore, your files and sessions can be shared, to some extent, across logins and machines.

However, your files are not automatically persist across sessions or reboots. To **avoid losing data**, please always place important files under `~/Desktop/myfiles/` (at the bottom right corner of the desktop)!

To synchrounize your files between the VDI machines and your own machine, you can either use online services (e.g., email or cloud storage) or let the VDI machines directly connect to your own machine (e.g., through `ssh`). Alternatively, you can also use the system clipboard to copy and paste small text between the VDI machines and your own machine.

Setting up Visual Studio Code

You are encouraged to use Visual Studio Code (VSCode) as the main integrated development environment (IDE) for the labs. You can open it by first clicking the `Show Applications` button at the bottom left corner of the desktop and then searching for `vscode` or clicking its icon in the last page.

The main UI of VSCode looks like this:

VSCode is highly extensive – i.e., it allows you to install various extensions to enhance its functionality. We encourage you to install the following extensions: 1. `Verilog-HDL/SystemVerilog/Bluespec SystemVerilog` for Verilog/SystemVerilog support. 2. `RISC-V Support` for RISC-V assembly support. 3. `cs200` for visualizing RTL simulation and RISC-V emulation for this course. 4. `MemoryView` for visualizing memory contents in RISC-V emulation.

Installing an extension

1. Click on the `Extensions` button on the left sidebar or press Ctrl+Shift+x.
2. Search for the extension by its name.
3. Install the extension by clicking the `Install` button.

Running RISC-V emulation with `cs200`

This course introduces the basics of computer architecture with RISC-V as the base instruction set architecture (ISA). You will learn how to write assembly

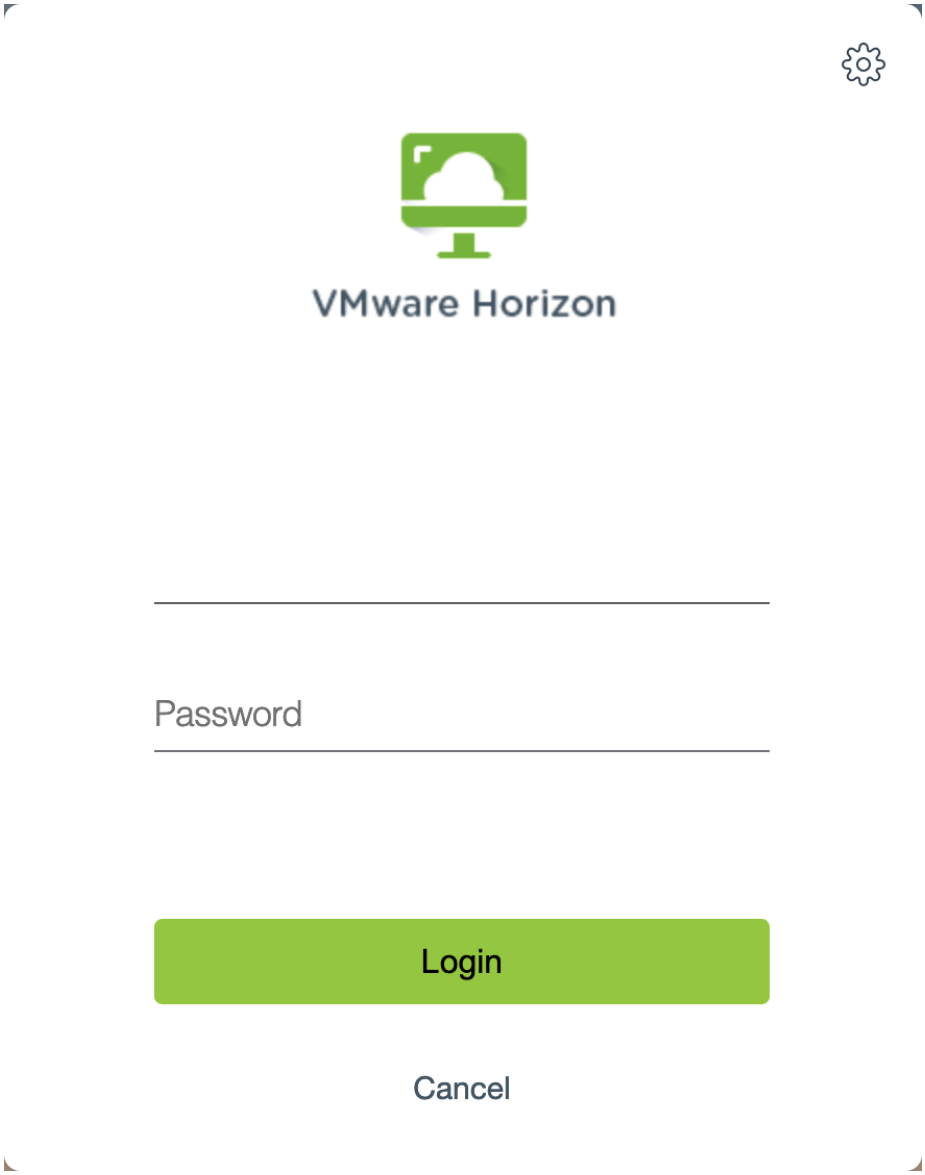

Figure 6: Logging in through the browser

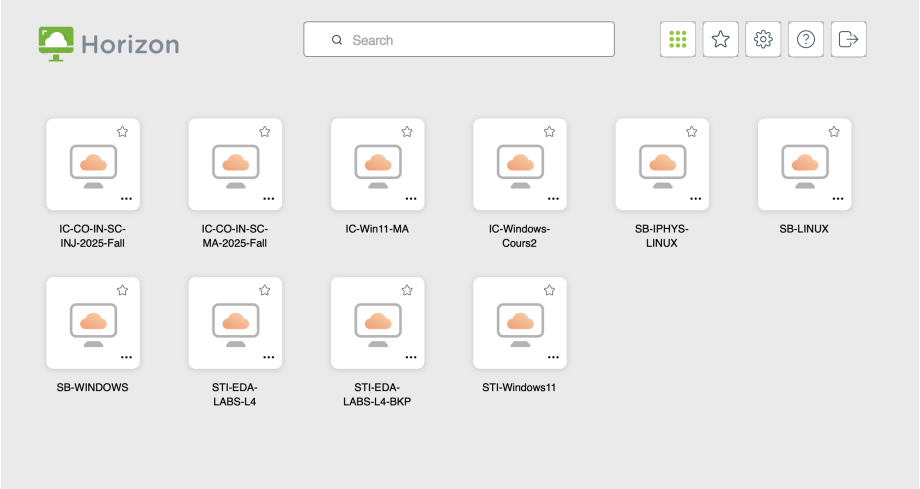

Figure 7: Main UI of the browser interface

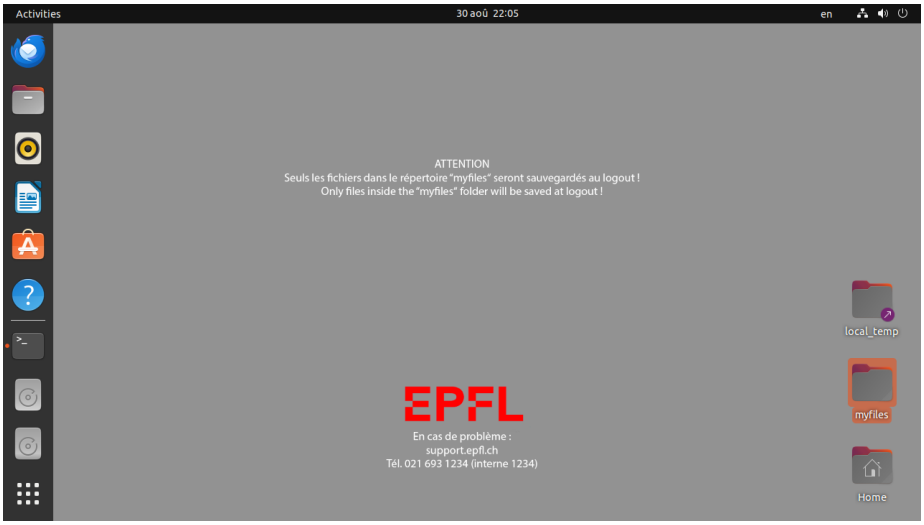

Figure 8: Desktop of the VDI machine

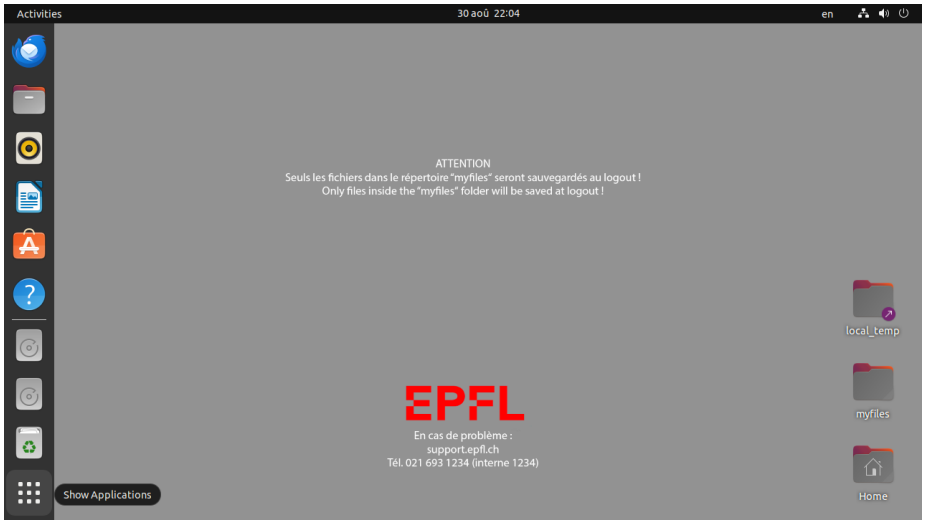

Figure 9: Show Applications

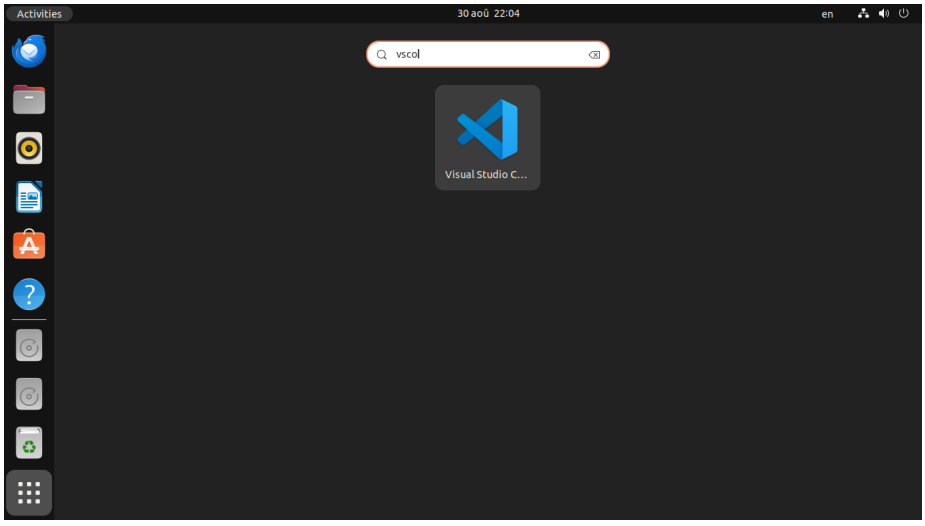

Figure 10: Searching for VSCode

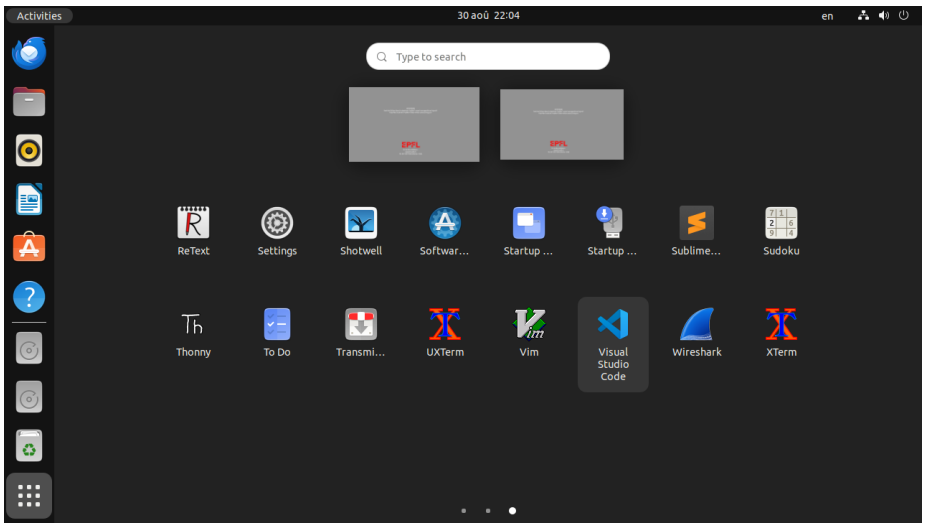

Figure 11: Browsing for VSCode

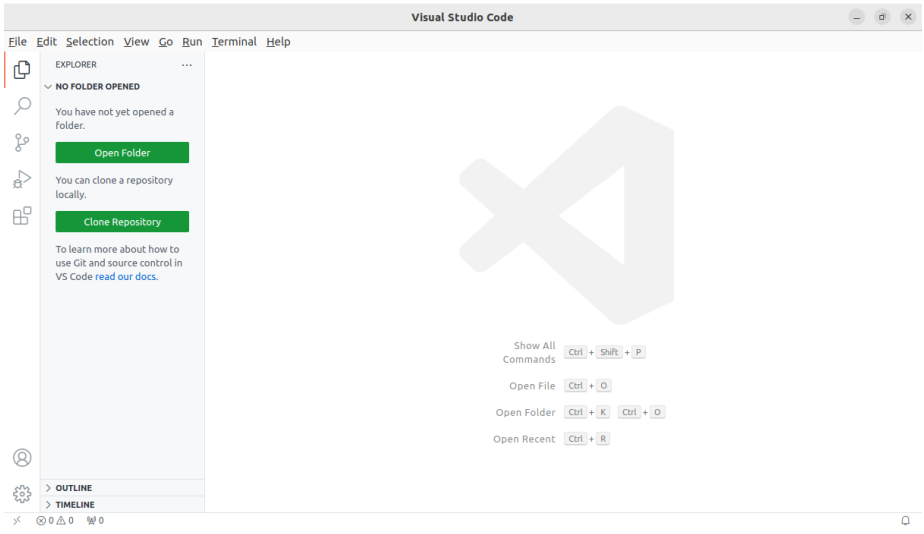

Figure 12: Main UI of VSCode

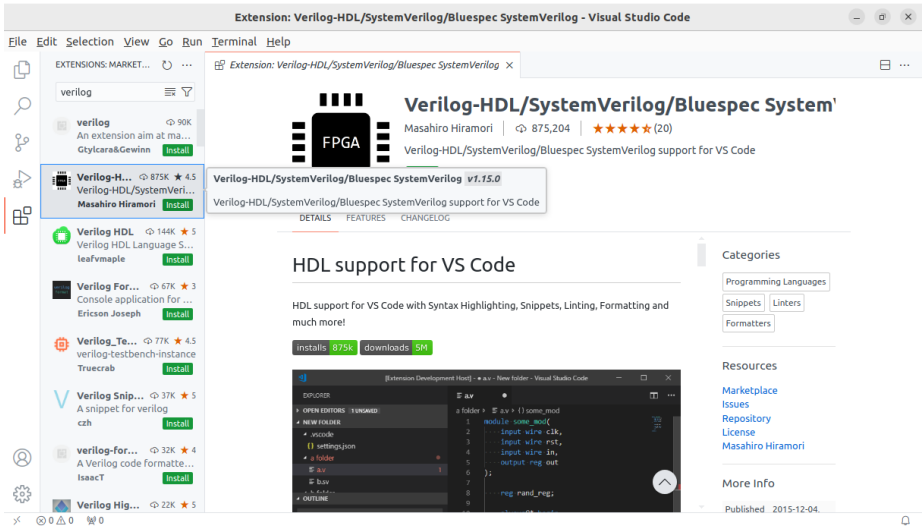

Figure 13: VSCode extensions

code for RISC-V, run the code in simulators, and finally design your own RISC-V-compatible CPU.

You may find the following resources useful while learning the RISC-V ISA:

- The latest official specification of the RISC-V ISA can be found here.
- The textbook An Introduction to Assembly Programming with RISC-V by Prof. Edson Borin covers many topics of RISC-V programming in a more introductory fashion. The free online version of the book can be found here.

The `cs200` extension provides a convenient way for you to debug your assembly code (and even RTL design) with a virtual interface identical to the Gecko 5 board.

To use the `cs200` extension, you need to first download a project template from here or from moodle with the name `Assembly-Language Project Template`, which contains a makefile and a linker script to compile your assembly code, and a pre-compiled RISC-V model for emulation. You need to extract the files to a directory with the following command and later put your assembly code (with file extension as `.s`) there:

```
tar xf /path/to/project.tar.gz
```

P.S. Do not forget to replace `/path/to/project.tar.gz` with the actual path to the downloaded file.

A small example

Let's start with a simple example to demonstrate the usage of the `cs200` extension. Consider the following RISC-V assembly code:

```
# test.s

.section ".text"

.global _start

_start:
  la   a0, 0x80001000
  la   a1, 0x80002000
  li   a2, 1

1:
  sb   a2, 0(a0)
  addi a0, a0, 1
  blt  a0, a1, 1b

2:
  j    2b
```

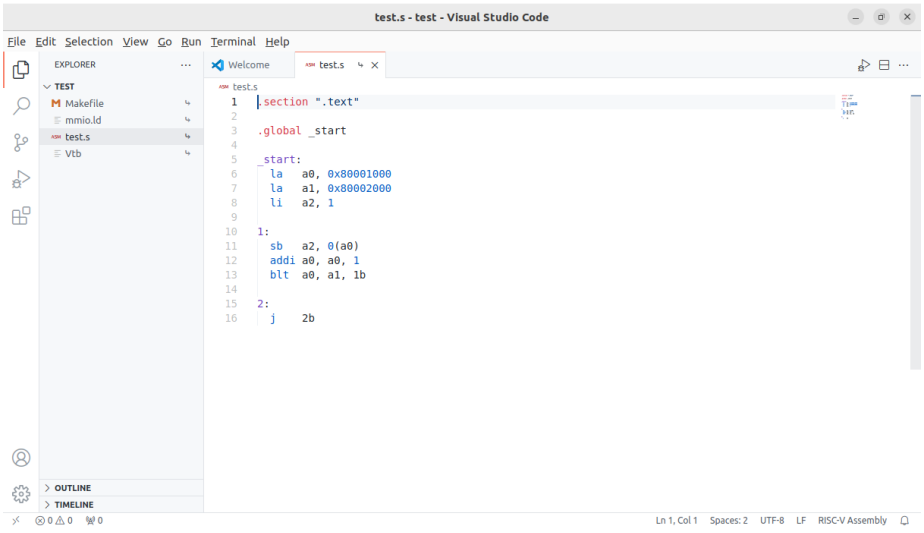

Figure 14: The assembly code

where the program uses the `sb` instruction to set every byte between `0x80001000` to `0x80002000` to 1 and then spins forever.

By clicking the `Debug File` button near the top right corner of the editor or

pressing F5, you can start the emulation of the code. The `cs200` extension will automatically compile the assembly code into a RISC-V binary and load it into our pre-compiled model. The VSCode UI will also switch into the debug mode as follows:

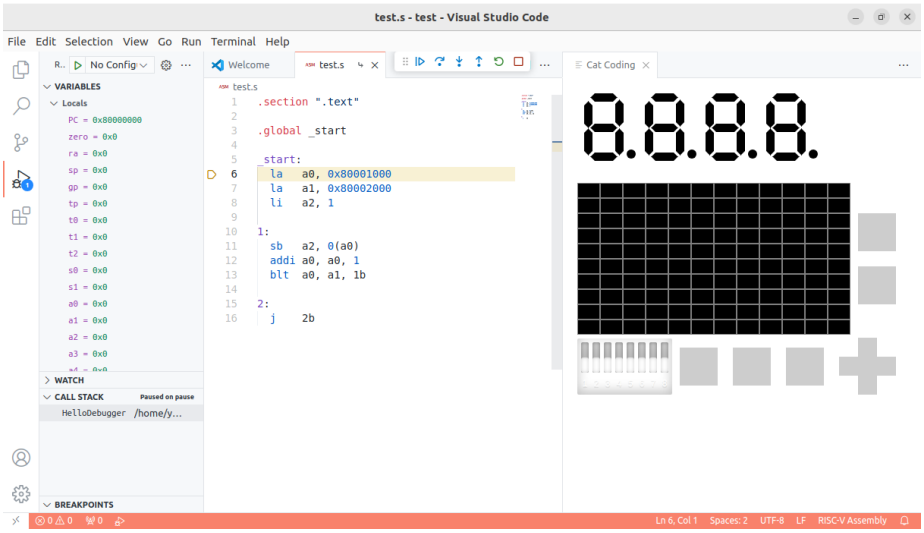

Figure 15: Debug GUI with virtual peripherals

where on the left panel the registers are displayed, while on the right panel the peripherals on the Gecko 5 board, i.e., the seven-segment displays, LEDs, switches, and buttons are shown. The assembly code is show in the center panel, with the line that contains the instruction to be executed next highlighted in yellow.

Similar to debugging normal programs, you can press the `Step Over` button or press F10 to execute one single instruction. You will observe that the highlighted line moves down as the program executes, and some registers in the left panel are also highlighted they are modified by the program.

Setting up memory view

You can also visualize the memory contents and updates with `MemoryView`. Please note that this extension is not maintained by our team and can have issues. The known common issues are discussed in the following section. You need to go over the following steps to enable it: 1. Open the command palatte by pressing Ctrl+P, enter `>open settings`, and select `Preferences: Open Workspace Settings (JSON)`.

2. In the opened `settings.json` file, add the following content to allow the `MemoryView` extension interacting with the `cs200` debugger:

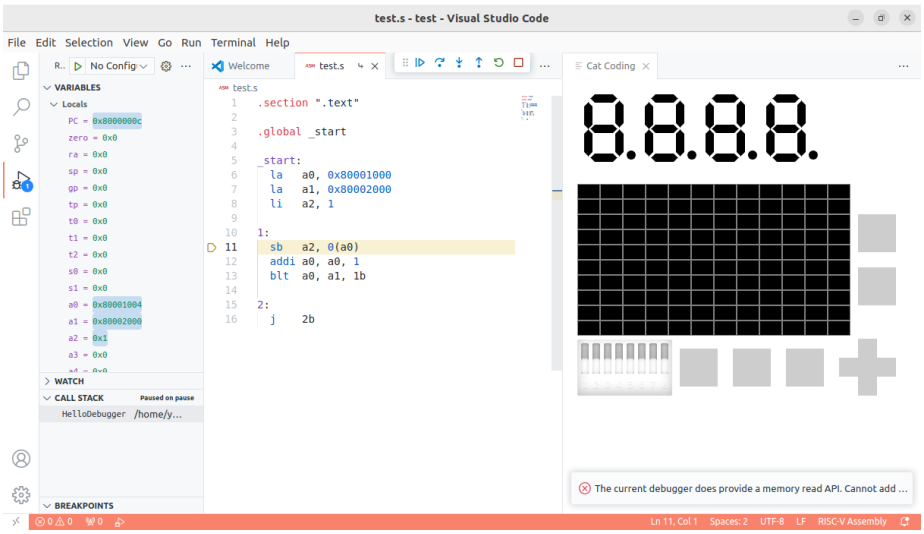

Figure 16: Continuing the debug

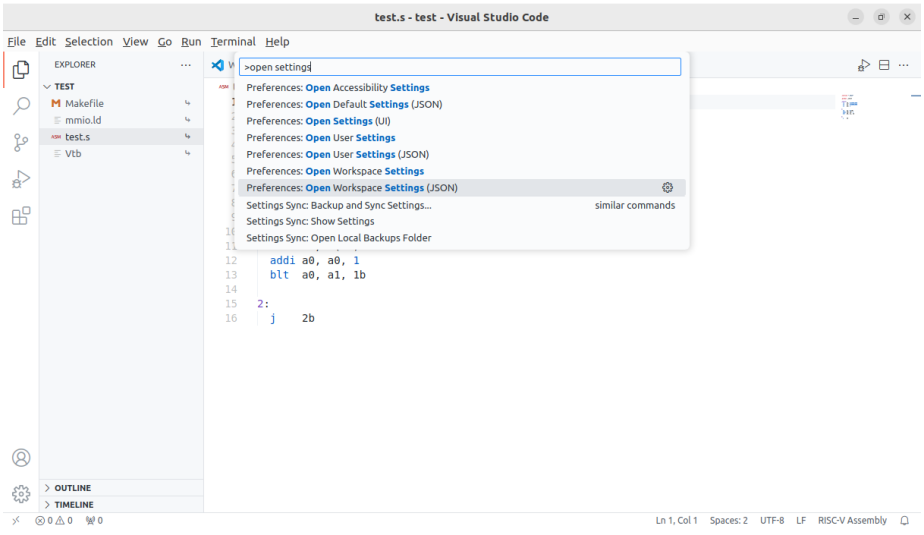

Figure 17: Opening workspace settings

```
{
    "memory-view.trackDebuggers": [
        "cs200"
    ]
}
```

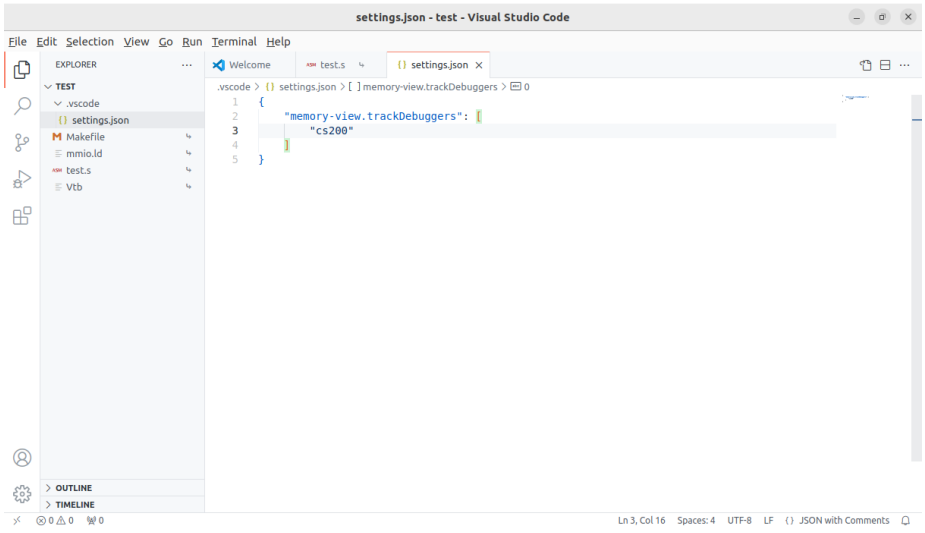

Figure 18: Modifying the settings

3. Start the debug by clicking the `Debug File` button or pressing F5.
4. Open the command palatte again, type `>memoryview`, and select `MemoryView: Add new memory view (for debugger)`.
5. Enter the base address of the memory you want to visualize. In our case, the updated memory starts at `0x80001000`. It will ask you the new memory view size, but it expects the same value as the base address, so you can just type `0x80001000` again.
6. The memory view will be shown as follows:

If there is any error occured, you can try to restart the debug session by clicking the `Restart` button or pressing Ctrl+Shift+F5 and then repeat from step 4.

By pressing F10 multiple times, you can observe that bytes starting from `0x80001000` are grudually updated to 1, corresponding to the program's behavior.

You can also create a breakpoint of an instruction by clicking the leftmost portion of the corresponding line as shown below:

The breakpoint will be shown as a red dot, and the program will stop executing when it reaches the breakpoint, i.e., when it is about to execute the corresponding instruction.

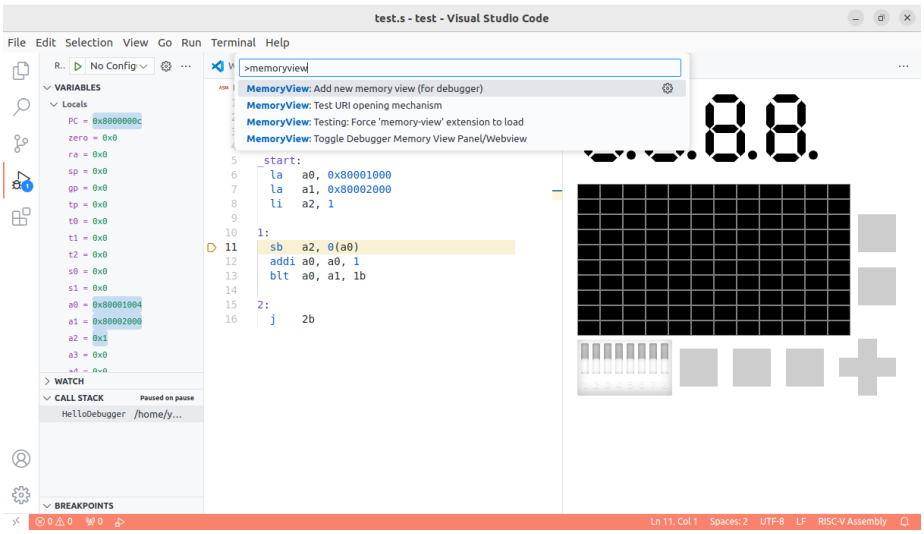

Figure 19: Attaching the memory view

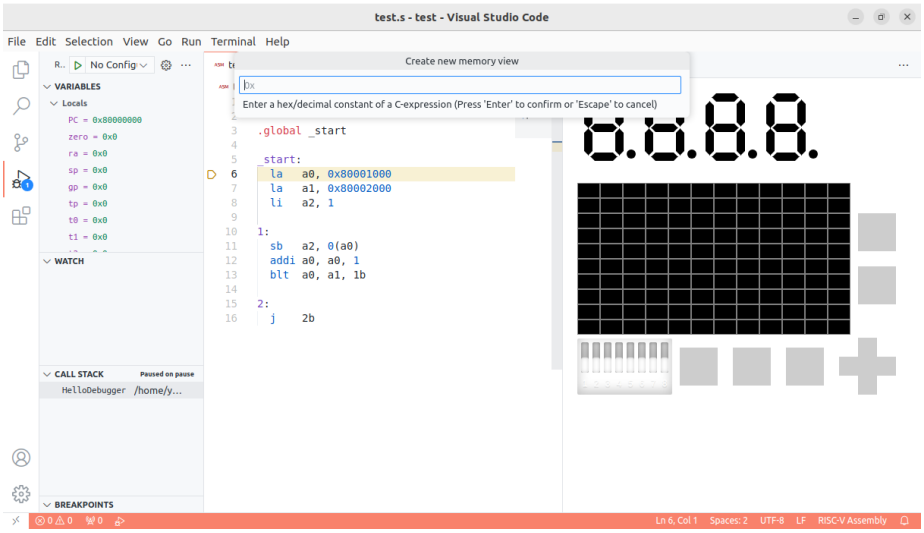

Figure 20: Specifying the base address for the memory view

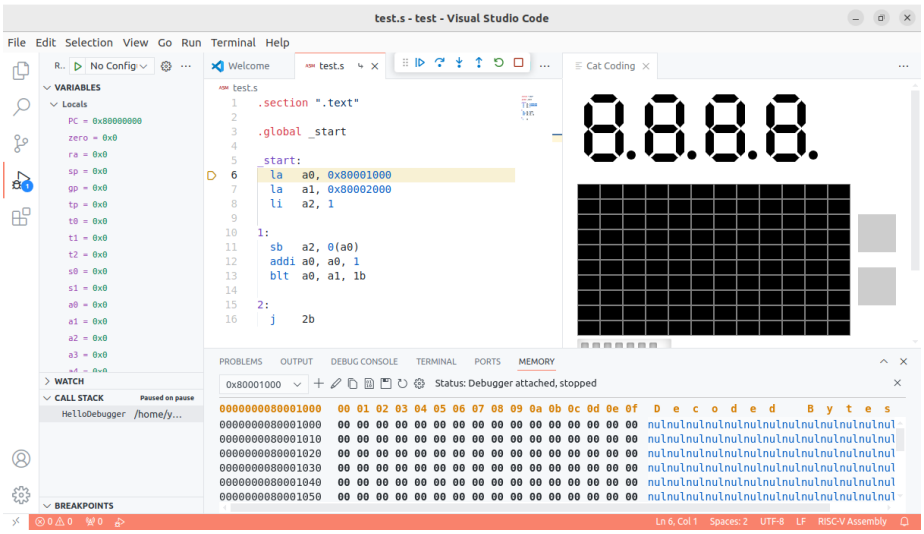

Figure 21: GUI of the memory view

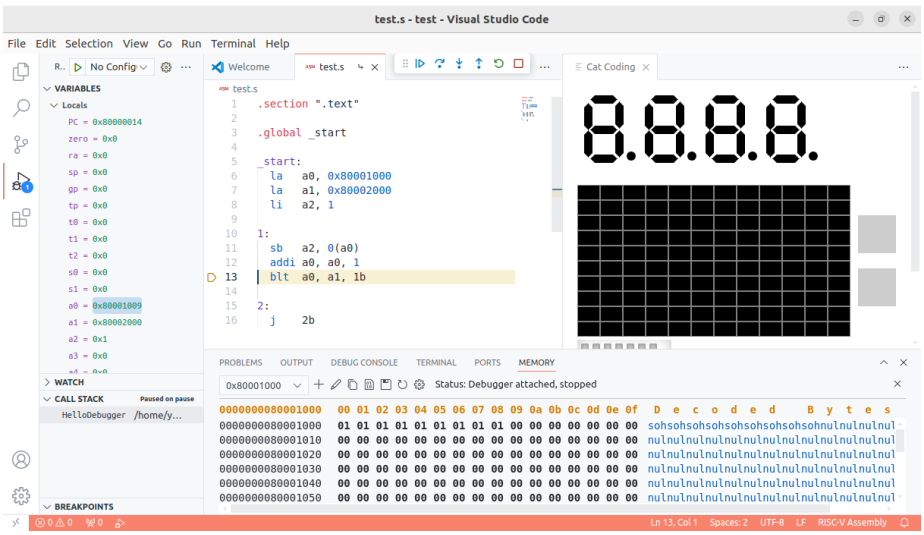

Figure 22: Memory is being updated

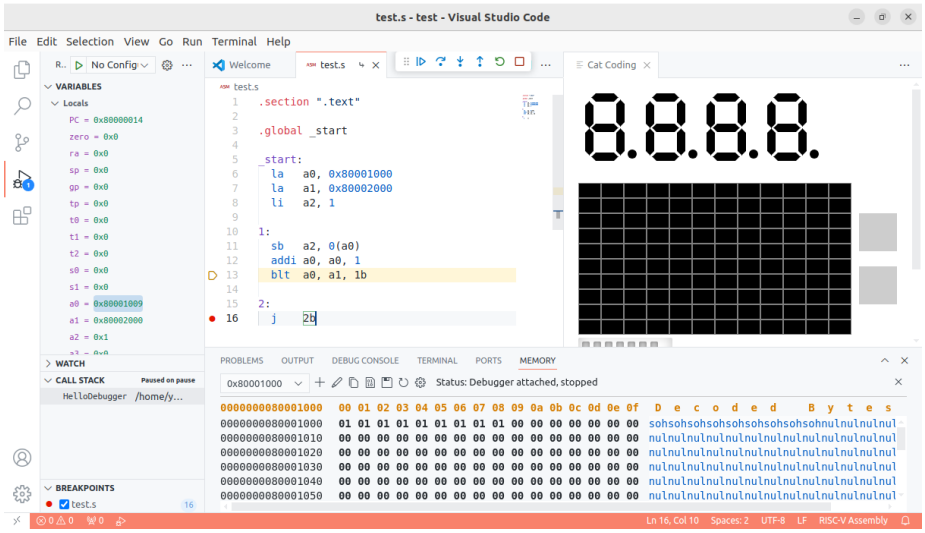

Figure 23: Setting a new breakpoint

After setting the breakpoint, you can resume executing the program and wait for hitting the breakpoint by clicking the `Continue` button or pressing F5. In our case, the execution will stop after it changes all the bytes from `0x80001000` to `0x80002000` to 1:

The CPU you write can also work with the `cs200` extension. However, you need to implement various SystemVerilog direct programming interface (DPI) functions to interact with the extension properly.

Known issues and solutions for the `memoryview` extension

The `memoryview` is an open source extension that is not officially maintained by our TA team. Therefore, there might be some issues when using it. Here are some common errors and solutions:

- Command `MemoryView: Add new memory view (for debugger)` resulted in an error. Command `mcu-debug.addMemoryView` not found

RISC-V Venus Simulator extension conflict with the memory-view extension and creates this bug. Uninstalling the RISC-V Venus Simulator extension and restarting the debug session usually fixes the issue.

- The current debugger does not provide a memory read API. Cannot add a memory view

This is an issue that the `memoryview` gives claiming that the debugger does not provide a memory read API. To fix this, you can open the command window by pressing Ctrl+P and typing `>Preferences: Open Workspace Settings`

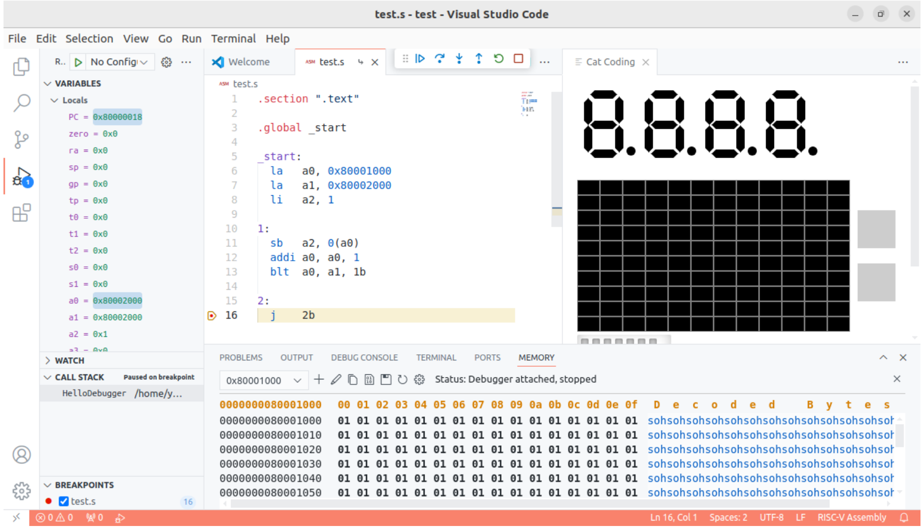

Figure 24: Hitting the breakpoint

(JSON) to open the settings. Add the following lines if they do not exist. Even if the line is already there, saving the file again and restarting the debug session usually fixes the issue.

```
{
    "memory-view.trackDebuggers": [
        "cs200"
    ]
}
```

- No session with the session id ... Probably a bug or debugger type that we are not tracking. Cannot add a memory view

This is another issue that the `memoryview` gives claiming that there is no session with the session id. To fix this, you can open the command window by pressing Ctrl+P and typing `>Preferences: Open Workspace Settings (JSON)` to open the settings. Add the following lines if they do not exist. Even if the line is already there, saving the file again and restarting the debug session usually fixes the issue.

```
{
    "memory-view.trackDebuggers": [
        "cs200"
    ]
}
```

Simulating and debugging RTL design

In this course, you will also learn how to design and simulate simple CPUs using Verilog, a high-level hardware description language.

A small example

For the following example, please save the Verilog files in a different folder than the one containing the assembly code, i.e. not in the root folder of the template project. Otherwise, `Vtb` might not work properly when you try to go back and run the assembly code.

Consider the following Verilog code that implements a module that can perform various logical operations on two 32-bit inputs and output the result one cycle later:

```
// lu.sv

module lu (
  input         clk_i,
  input         rst_i,
  input  [31:0] a_i,
  input  [31:0] b_i,
  input  [ 1:0] sel_i,
  output [31:0] out_o
);

  // decode the `sel_i` signal
  wire sel_and_w = (sel_i == 2'b00);
  wire sel_or_w  = (sel_i == 2'b01);
  wire sel_xor_w = (sel_i == 2'b10);

  // perform the logical operations
  wire [31:0] res_and_w = a_i & b_i;
  wire [31:0] res_or_w  = a_i | b_i;
  wire [31:0] res_xor_w = a_i ^ b_i;

  // generate the final result according to the selection
  wire [31:0] res_w = {32{sel_and_w}} & res_and_w |
                      {32{sel_or_w }} & res_or_w  |
                      {32{sel_xor_w}} & res_xor_w;

  // delay the output by one cycle
  reg  [31:0] out_r;

  always @(posedge clk_i)
    if (rst_i)
```

```
      out_r <= 32'b0;
    else
      out_r <= res_w;

  // output
  assign out_o = out_r;

endmodule
```

You are encouraged to stick to the CS-200 Verilog Coding Style Guide when writing Verilog code. You can also use Verible to perform linting on your Verilog code.

You can download Verible from here. It's important that you download the correct version for your machine. If you're using the VM, it should be for Linux and `x86-64` architecture. You need to decompress the downloaded file:

```
tar xf verible-[version]-linux-static-x86_64.tar.gz
```

and add the `bin` directory to your `PATH` environment variable:

```
export PATH=$PATH:/path/to/verible-[version]/bin
```

Then, you can lint your Verilog code by running the following command:

```
verible-verilog-lint [file] --rules_config flags.txt
```

where `[file]` is the Verilog file you want to lint and `flags.txt` contains the rules you want to apply with the following content:

```
-always-comb
+port-name-suffix
signal-name-style="style_regex:[a-z_0-9]+"
-explicit-parameter-storage-type
+parameter-name-style="localparam_style:ALL_CAPS;parameter_style:ALL_CAPS"
```

Compiling RTL design using verilator

We use verilator to compile the Verilog code into a C++ model, which can later be compiled to an executable to simulate the design.

To simulate your design, you need a test bench to generate stimulus to it, i.e., driving the input signals of your design with desired values. In addition, you can also optionally check the output signals to see if they are correct. A minimum test bench for the logical unit above can be as follows:

```
// tb.sv

module tb ();

  // generate clock and reset
```

```
reg  clk_r;
reg  rst_r;

// period: 1 [time unit]
always #0.5 clk_r = ~clk_r;

initial begin
  clk_r = 1'b1;
  rst_r = 1'b1;

  // reset asserts for 20 [time units]
  #20
  rst_r = 1'b0;

  // after 1000 [time units], the simulation stops
  #1000
  $finish();
end

// randomized inputs
reg  [31:0] a_r;
reg  [31:0] b_r;
reg  [31:0] sel_r;

always @(posedge clk_r) begin
  a_r   <= $urandom();
  b_r   <= $urandom();
  sel_r <= $urandom();
end

// instantiation
wire [ 1:0] sel_w = sel_r[1:0];
wire [31:0] out_w;

lu lu_0_(
  .clk_i (clk_r),
  .rst_i (rst_r),
  .a_i   (a_r  ),
  .b_i   (b_r  ),
  .sel_i (sel_w),
  .out_o (out_w)
);

// dump waveform for all signals in `tb`
initial begin
    $dumpfile("dump.vcd");
```

```
    $dumpvars(0, tb);
  end

endmodule
```

Then you can invoke verilator with the following arguments to compile Verilog into C++ and then build an executable:

```
verilator --timescale 1ns/1ns --top-module tb \
--cc --exe --binary --timing --trace --trace-underscore -O2 lu.sv tb.sv
```

Figure 25: Compiling the design using verilator

which will later generate the executable model as `obj_dir/Vtb`. You can then start the simulation by simply running it in the terminal with the following command:

```
./obj_dir/Vtb
```

You can refer to the Verilator User's Guide for more advanced usage of verilator.

Using GTKWave

The compiled model, when executing, can generate a waveform file that contains the values of all the signals in the design at each time step. We use GTKWave to visualize the waveform and help us debug the design.

Under the simulation directory, you can invoke GTKWave as follows:

```
gtkwave dump.vcd
```

where `dump.vcd` is the waveform file generated by the simulation.

Figure 26: Running the compiled model

The GTKWave GUI looks as follows:

In the left panel, you can see the signal search tree (SST) that shows the hierarchy of the design. You can expand each module to see all its submodules. When a module is selected, all its signals will be shown in the below list:

You can also enter the signal name into the input box at the bottom left corner to filter signals. Only signals whose name starts with the input string will be shown:

By double clicking on or dragging a signal, you can add it to the waveform view in the right panel, which shows values of each added signal at each time step:

In the `Waves` view, you can use the following mouse/keyboard shortcuts to easily browse the waveform: - ScrollUp: Scroll left (towards smaller simulation time) - ScrollDown: Scroll right (towards larger simulation time) - Shift+ScrollUp: Slowly scroll left - Shift+ScrollDown: Slowly scroll right - Ctrl+ScrollUp: Zoom in the view (with smaller simulation time range) - Ctrl+ScrollDown: Zoom out the view (with larger simulation time range)

For more advanced usage, please refer to the GtkWave documentation.

Debugging RTL design with `cs200`

The `cs200` extension also supports debugging RTL designs. Some of the functionality of the `cs200` will not be available depending on what is implemented in the RTL design. The extension utilizes the SystemVerilog Dynamic Programming Interface (DPI-C) functions in the design. You are not expected to implement

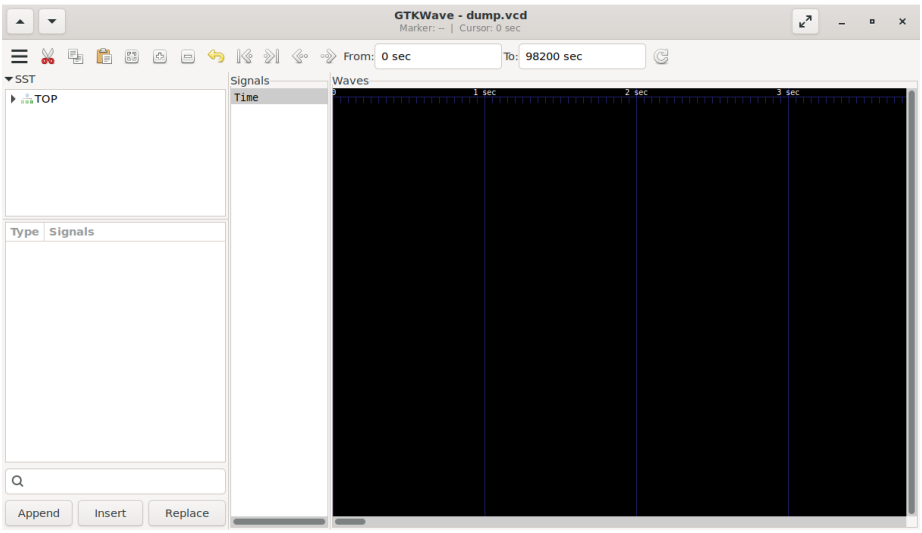

Figure 27: Main UI of GTKWave

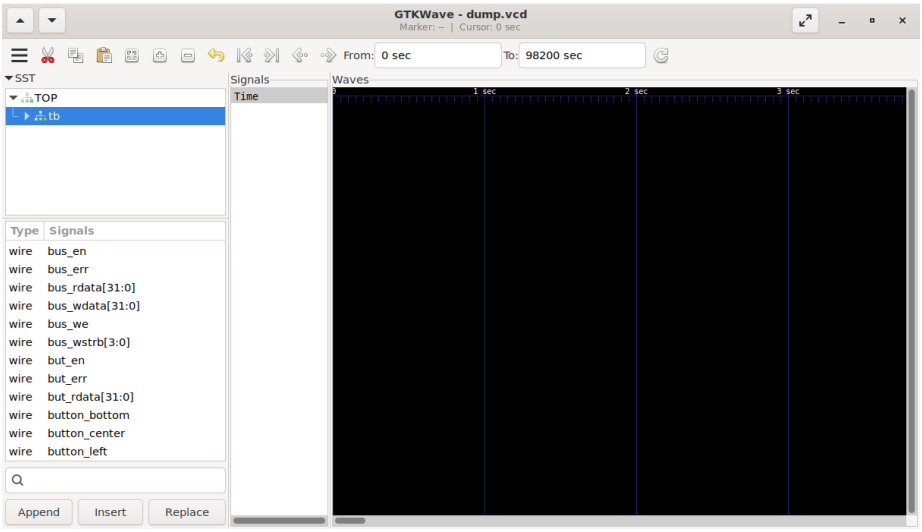

Figure 28: Signal search tree

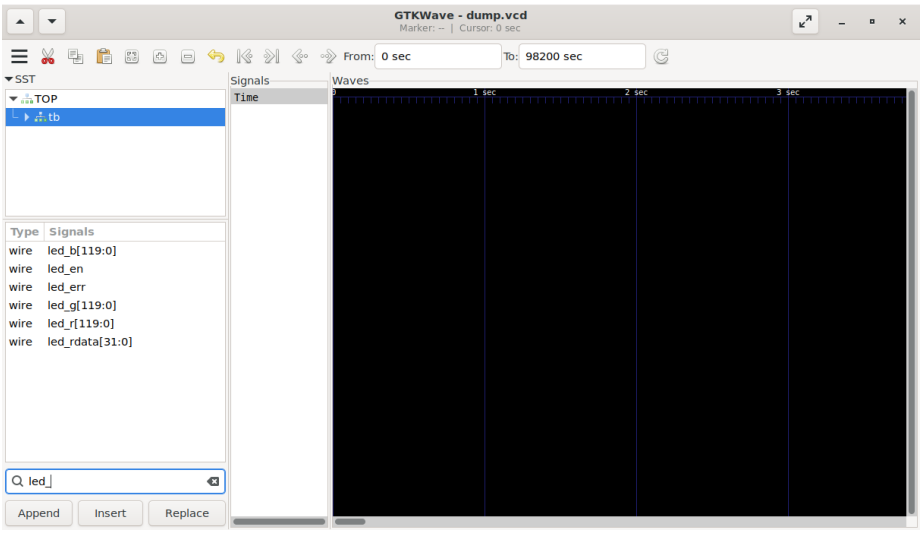

Figure 29: Filtering out signals

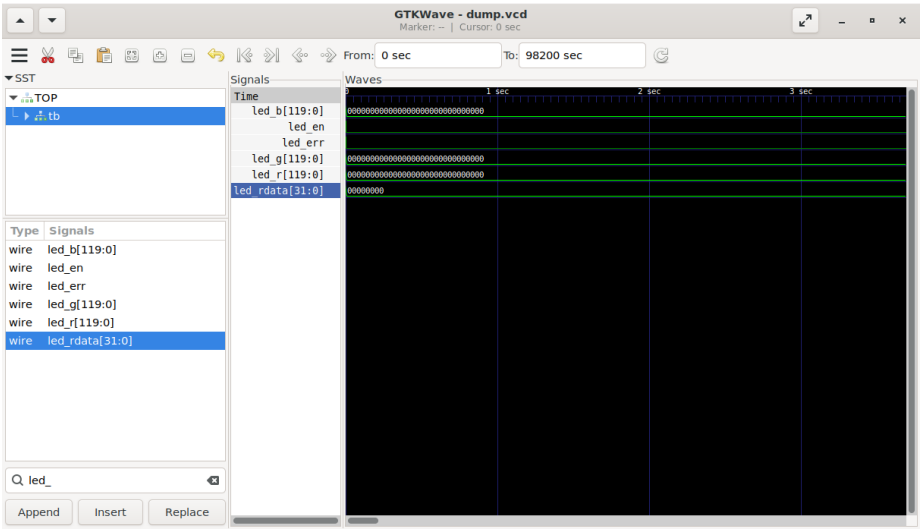

Figure 30: Adding signals to the waveform

or know anything about these functions during the course. For example, the left panel will not show the registers if the design does not have the necessary DPI-C functions to interact with the extension.

The right panel will show the peripherals on the Gecko 5 board, i.e., the seven-segment displays, LEDs, switches, and buttons regardless of the RTL design. If you would like to interact with the peripherals, you need to have proper input and output signals in your top level RTL design. The proper names and signal widths are given in the following snippet. Every signal except `clk_i` and `rst_ni` are optional, and you can choose to implement only the peripherals you need.

```
module tb (
        input        clk_i,
        input        rst_ni,
        output reg [7:0] sevensegment_1_o,
        output reg [7:0] sevensegment_2_o,
        output reg [7:0] sevensegment_3_o,
        output reg [7:0] sevensegment_4_o,
        output reg [119:0] led_r_o,
        output reg [119:0] led_g_o,
        output reg [119:0] led_b_o,
        input button_top_i,
        input button_bottom_i,
        input button_left_i,
        input button_right_i,
        input button_center_i,
        input [7:0] dip_switches_i,
        input joystick_up_i,
        input joystick_down_i,
        input joystick_left_i,
        input joystick_right_i,
        input joystick_pressed_i
    );
```

An example of the flow to debug an RTL design is as follows:

1. Download the proper folder structure from this link.
2. Write the RTL design and press debug button in the top right corner.
3. The extension will compile the design and start the simulation.
4. The simulation will start, and you can interact with the peripherals on the right panel. You need to press the `Step Over` button or press F10 to execute one clock cycle.

Trace Option

The `cs200` extension also supports the trace option for RTL designs. The trace option will generate a `.vcd` file that can be opened with GTKWave at the end of

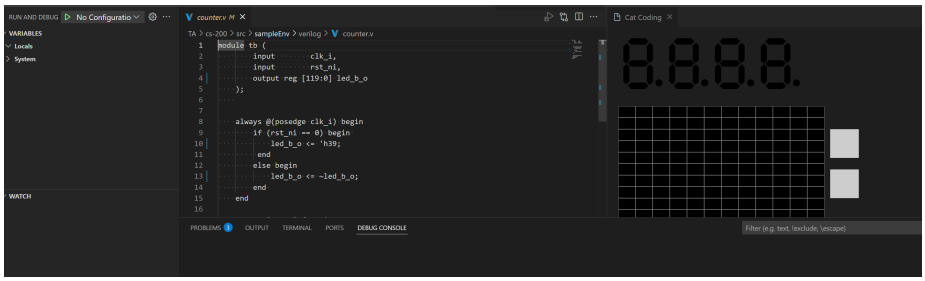

Figure 31: Debugging RTL design

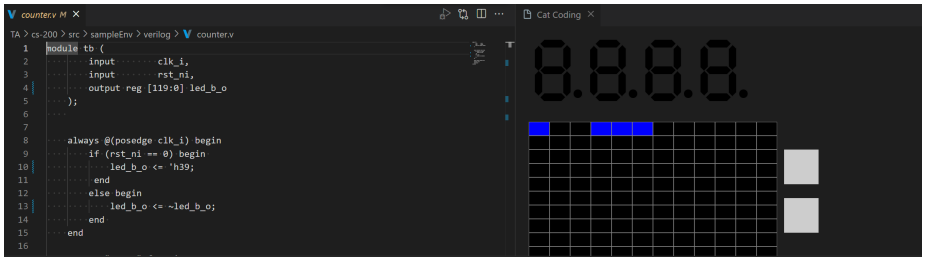

Figure 32: Debugging RTL design

the simulation. The trace option can be enabled in the settings of the extension. Please refer to the image below to enable the trace option.

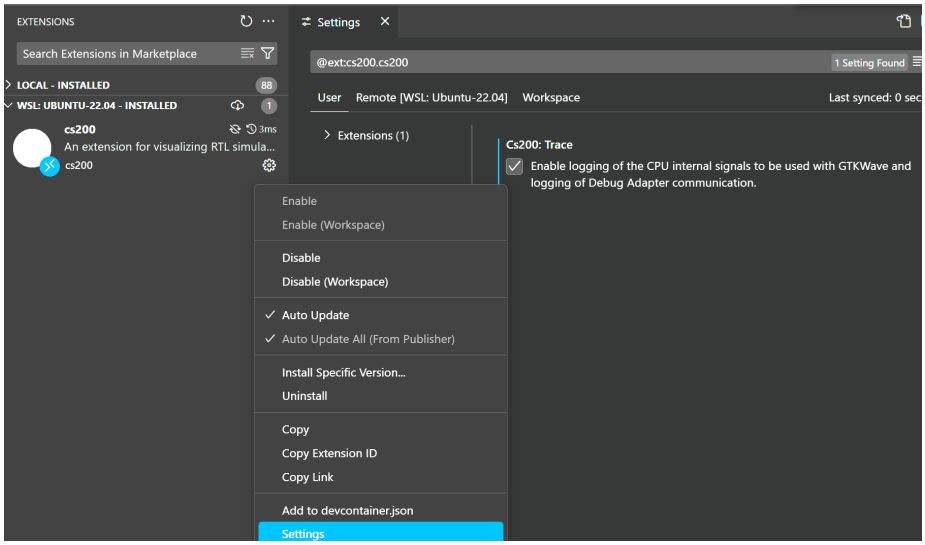

Figure 33: Trace Option