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. You can also work on your own machines, but there is no guarantee that the teaching staff can help you resolve issues specific to your setup.

Using physical machines in INF 3

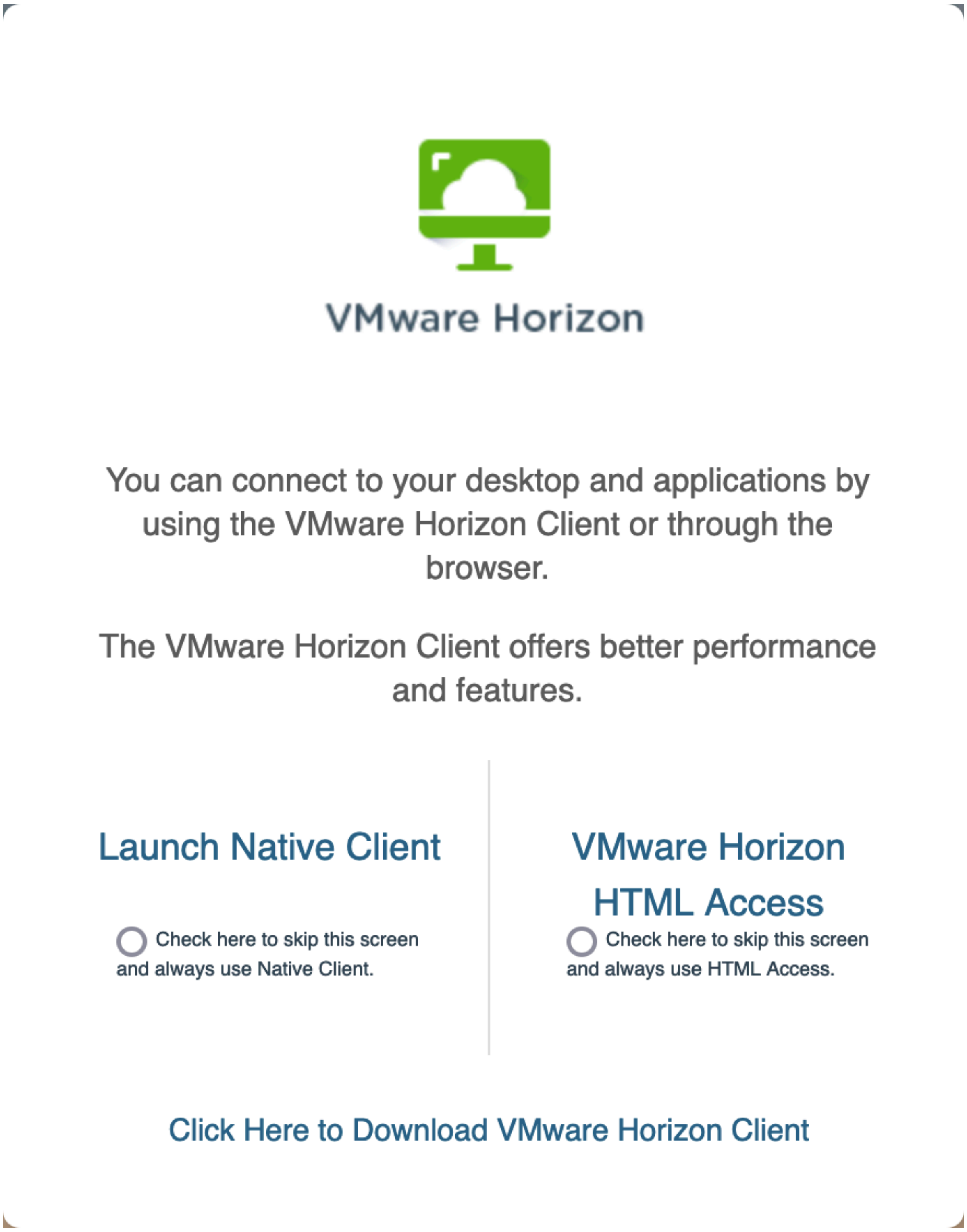
1. Make your way to INF 3.
2. Log into any available desktop using your GASPAR credentials.

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 `Click Here to Download VMware Horizon Client` at the bottom to download the installer (this page may vary across operating systems).

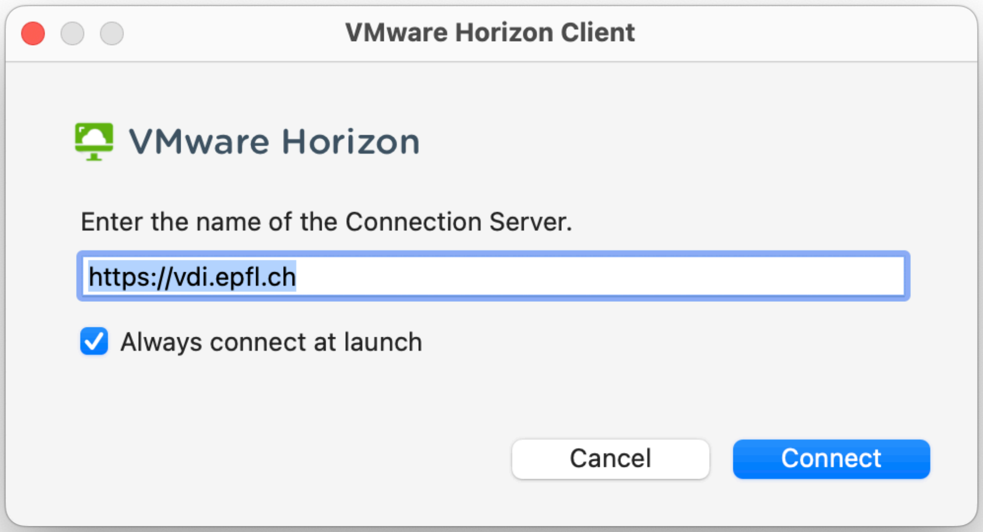

VMware Horizon

You can connect to your desktop and applications by using the VMware Horizon Client or through the browser.

The VMware Horizon Client offers better performance and features.

Launch Native Client
Check here to skip this screen and always use Native Client.
VMware Horizon HTML Access
Check here to skip this screen and always use HTML Access.
Click Here to Download VMware Horizon Client

3. Run the installer to install VMware Horizon Client on your machine.
4. Launch VMware Horizon Client (also named `vmware-view` in some platforms).
5. Enter `https://vdi.epfl.ch` as the connection server.

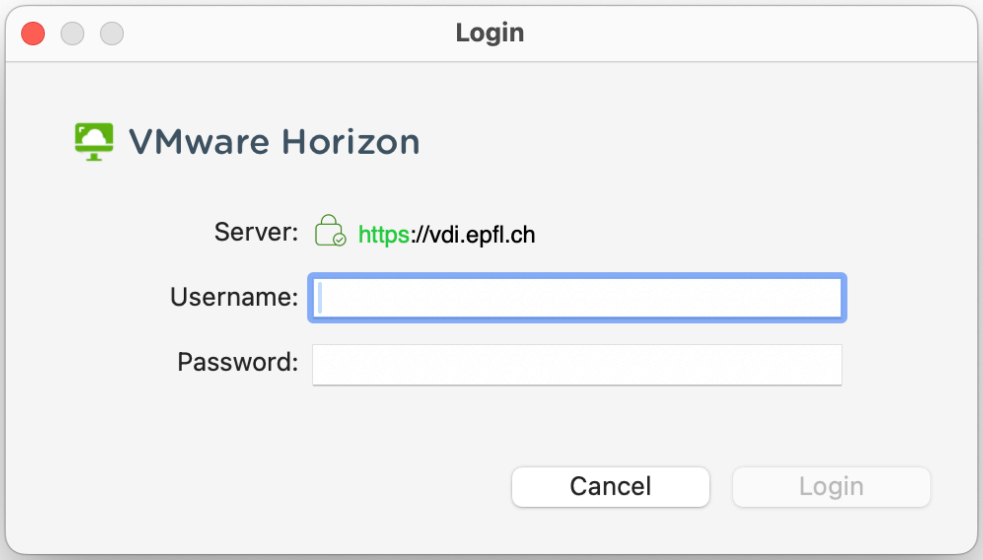


Enter the name of the Connection Server.

https://vdi.epfl.ch

✓ Always connect at launch

6. Log in with your GASPAR credentials.

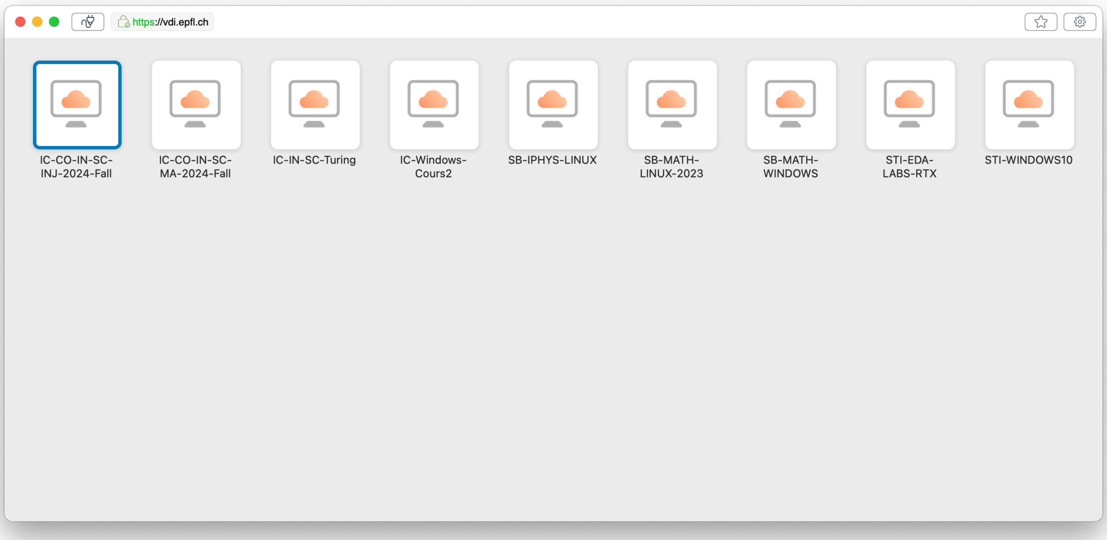


Server: https://vdi.epfl.ch

Username:

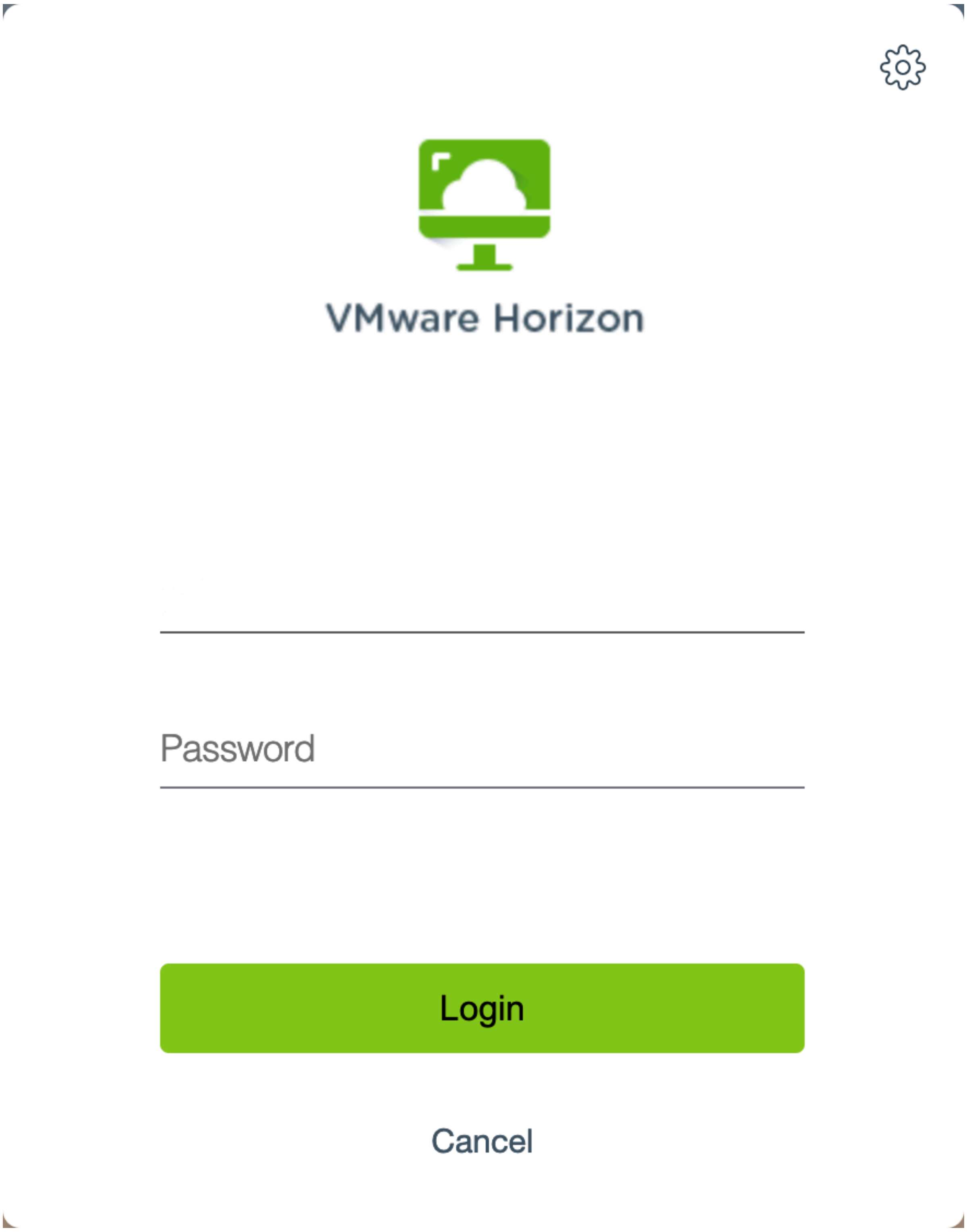
Password:

7. Select and connect to the `IC-CO-IN-SC-INJ-2024-fall` or `IC-CO-IN-SC-MA-2024-fall` machine.

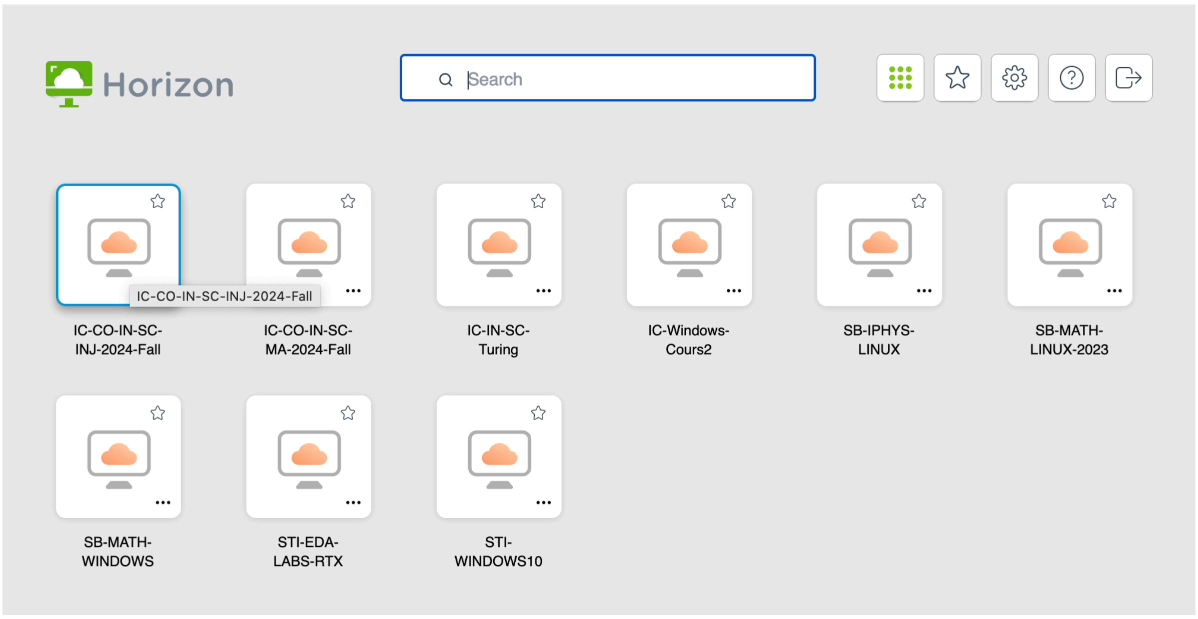


Web Browser

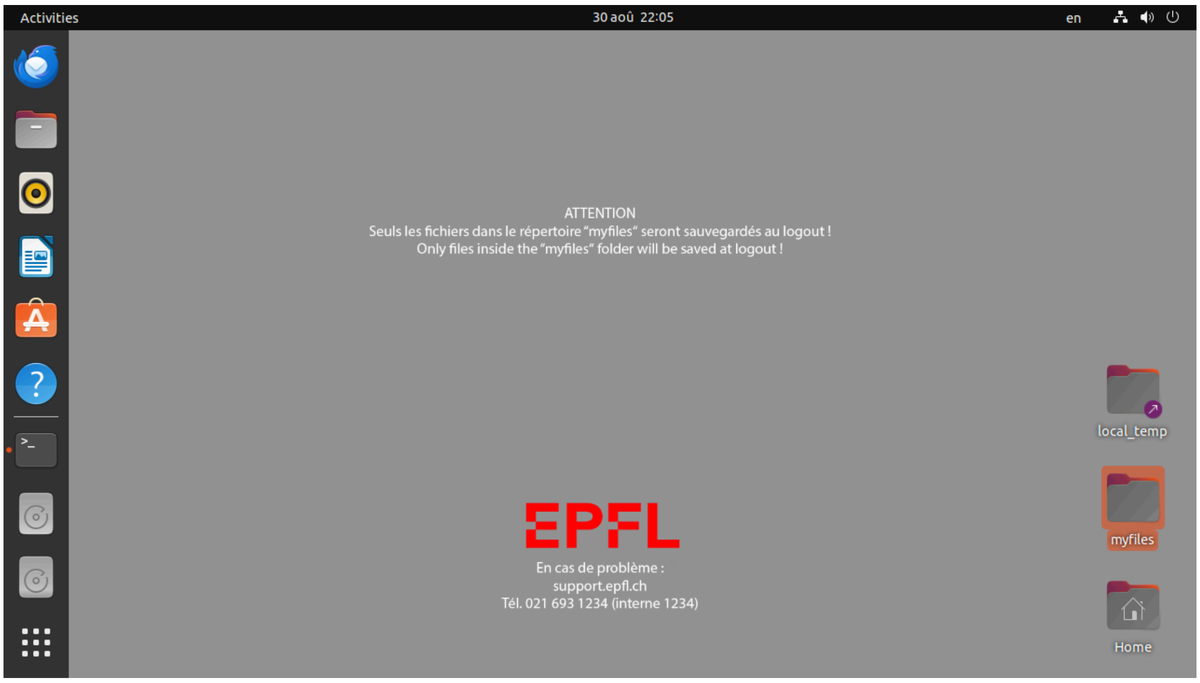
1. Navigate to https://vdi.epfl.ch.
2. Click on `VMware Horizon HTML Access`.
3. Log in with your GASPAR credentials.



4. Select and connect to the `IC-CO-IN-SC-INJ-2024-fall` or `IC-CO-IN-SC-MA-2024-fall` machine.



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



ATTENTION
Seuls les fichiers dans le répertoire "myfiles" seront sauvegardés au logout !
Only files inside the "myfiles" folder will be saved at logout !

En cas de problème :
support.epfl.ch
Tél. 021 693 1234 (interne 1234)

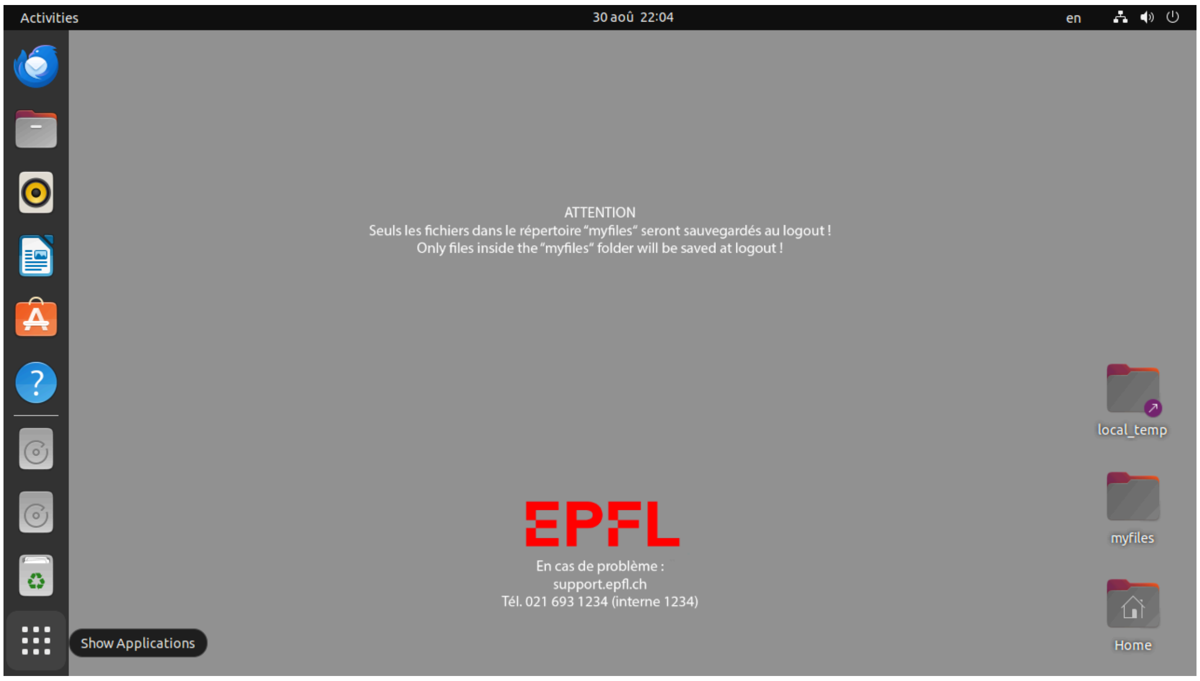
Under the hood, the physical machines in INF 3 are actually connecting to the same remote desktops as you would via the VDI. Therefore, your files and sessions can be shared, to some extent, across logins and machines.

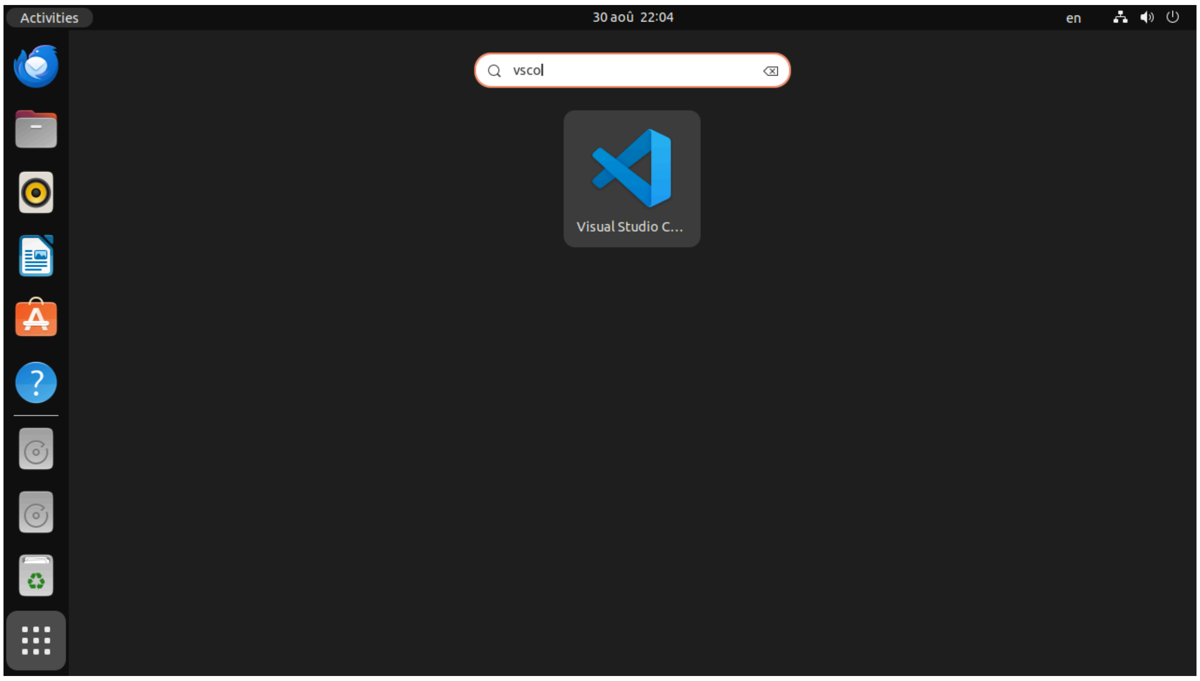
However, your files are not automatically persist across sessions or reboots. To **avoid losing data**, please always place important files under `~/Desktop/myfiles/` (at the bottom right corner of the desktop)!

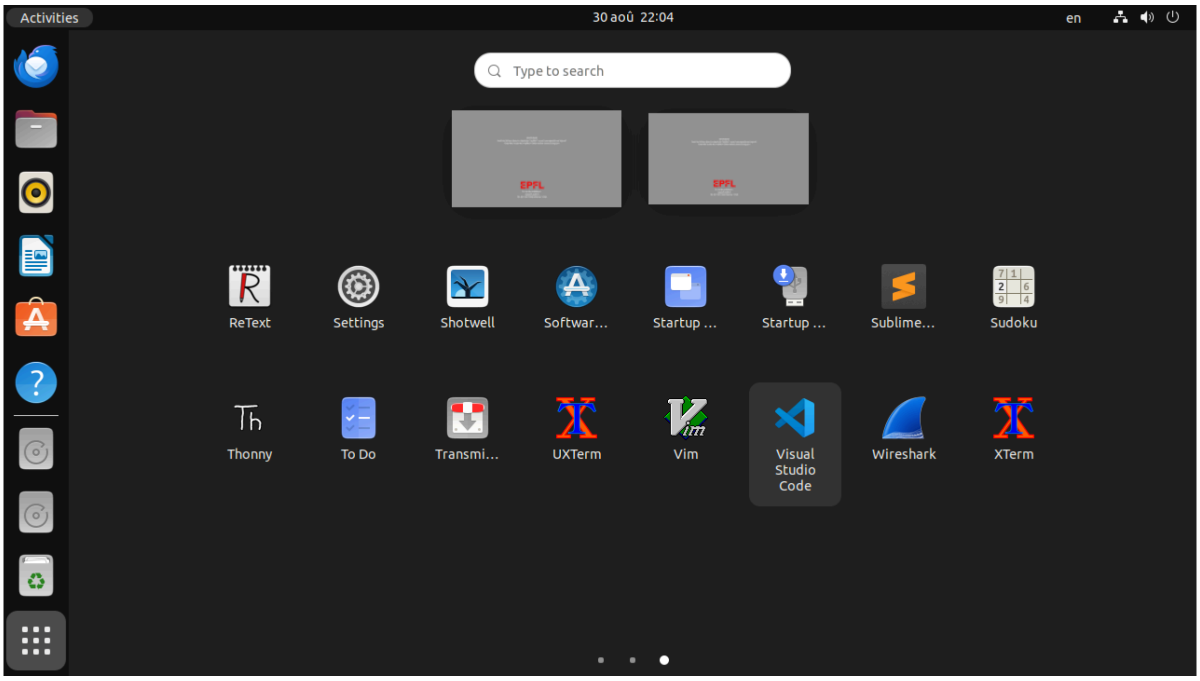
To synchrounize your files between the VDI machines and your own machine, you can either use online services (e.g., email or cloud storage) or let the VDI machines directly connect to your own machine (e.g., through `ssh`). Alternatively, you can also use the system clipboard to copy and paste small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

Setting up Visual Studio Code

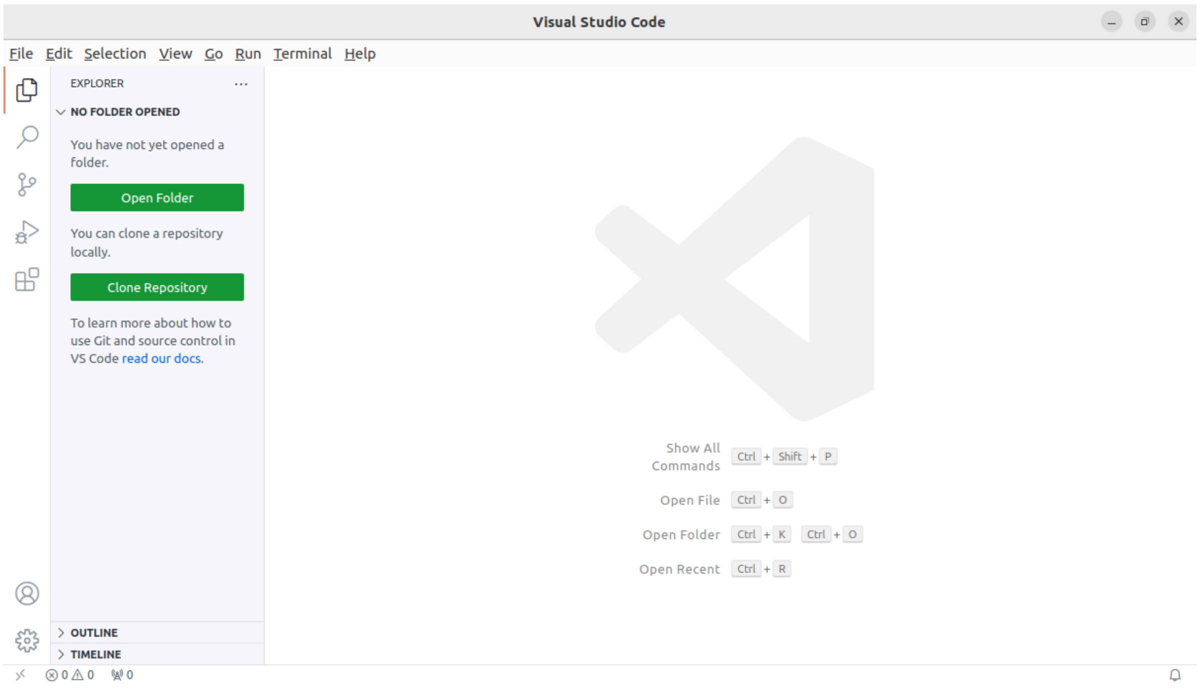
You are encouraged to use Visual Studio Code (VSCode) as the main integrated development environment (IDE) for the labs. You can open it by first clicking the `Show Applications` button at the bottom left corner of the desktop and then searching for `vscode` or clicking its icon in the last page.


Activities
30 aoû 22:04
en
ATTENTION
Seuls les fichiers dans le répertoire "myfiles" seront sauvegardés au logout !
Only files inside the "myfiles" folder will be saved at logout !
local_temp
myfiles
EPFL
En cas de problème :
support.epfl.ch
Tél. 021 693 1234 (interne 1234)
Show Applications
Home





The main UI of VSCode looks like this:



You have not yet opened a folder.

You can clone a repository locally.

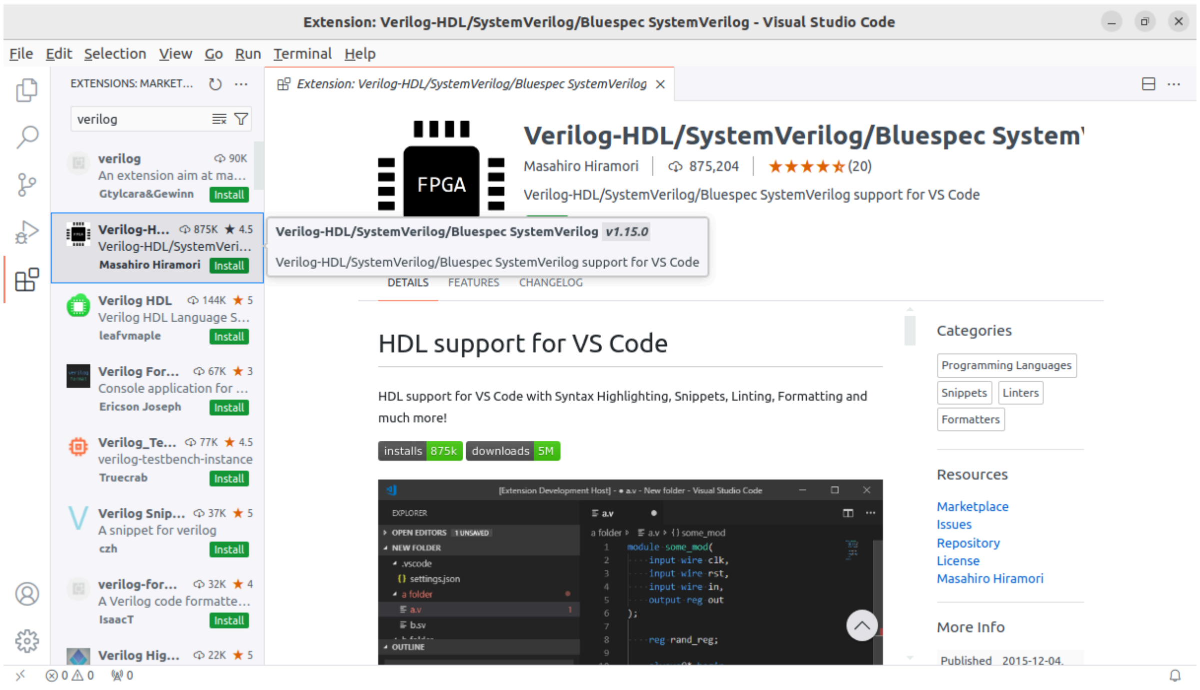
To learn more about how to use Git and source control in VS Code read our docs.

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.


Extension: Verilog-HDL/SystemVerilog/Bluespec SystemVerilog - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXTENSIONS: MARKET...
verilog
verilog 90K An extension aim at ma... Gtylcara&Gewinn Install
Verilog-H... 875K ★ 4.5 Verilog-HDL/SystemVeri... Masahiro Hiramori Install
Verilog HDL 144K ★ 5 Verilog HDL Language S... leafvmaple Install
Verilog For... 67K ★ 3 Console application for ... Ericson Joseph Install
Verilog_Te... 77K ★ 4.5 verilog-testbench-instance Truecrab Install
Verilog Snip... 37K ★ 5 A snippet for verilog czh Install
verilog-for... 32K ★ 4 A Verilog code formatte... IsaacT Install
Verilog Hig... 22K ★ 5
Extension: Verilog-HDL/SystemVerilog/Bluespec SystemVerilog
FPGA
Verilog-HDL/SystemVerilog/Bluespec System
Masahiro Hiramori | 875,204 | ★★★★★ (20)
Verilog-HDL/SystemVerilog/Bluespec SystemVerilog support for VS Code
Verilog-HDL/SystemVerilog/Bluespec SystemVerilog v1.15.0
Verilog-HDL/SystemVerilog/Bluespec SystemVerilog support for VS Code
DETAILS FEATURES CHANGELOG
HDL support for VS Code
installs 875k downloads 5M
Categories
Programming Languages
Snippets Linters
Formatters
Resources
Marketplace
Issues
Repository
License
Masahiro Hiramori
More Info
Published 2015-12-04
⊗0 ⚠0 0

HDL support for VS Code with Syntax Highlighting, Snippets, Linting, Formatting and much more!

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 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, 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:

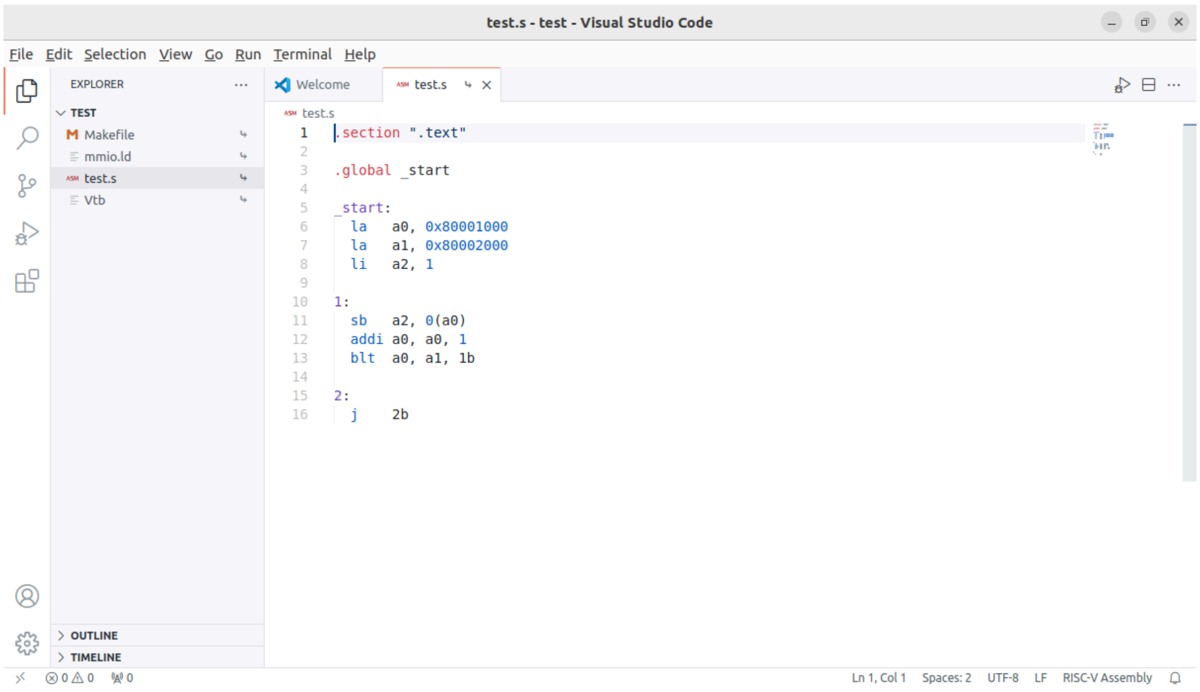
```
tar xf /path/to/project.tar.xz
```

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
```

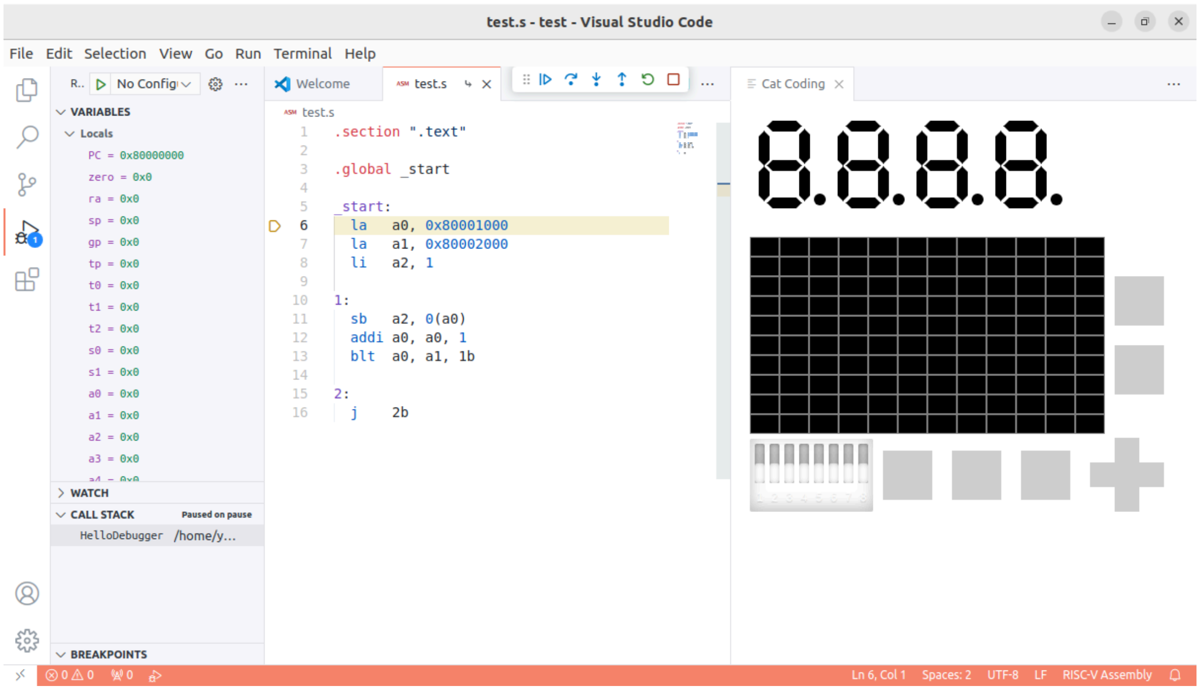
```
_start:
  la   a0, 0x80001000
  la   a1, 0x80002000
  li   a2, 1

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

2:
  j    2b
```


test.s - test - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
TEST
Makefile
mmio.ld
test.s
Vtb
Welcome
test.s
OUTLINE
TIMELINE
Ln 1, Col 1 Spaces: 2 UTF-8 LF RISC-V Assembly

```
.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
```

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:


test.s - test - Visual Studio Code
File Edit Selection View Go Run Terminal Help
No Config
Welcome
test.s
Cat Coding
VARIABLES
Locals
WATCH
CALL STACK
Paused on pause
BREAKPOINTS
Ln 6, Col 1 Spaces: 2 UTF-8 LF RISC-V Assembly

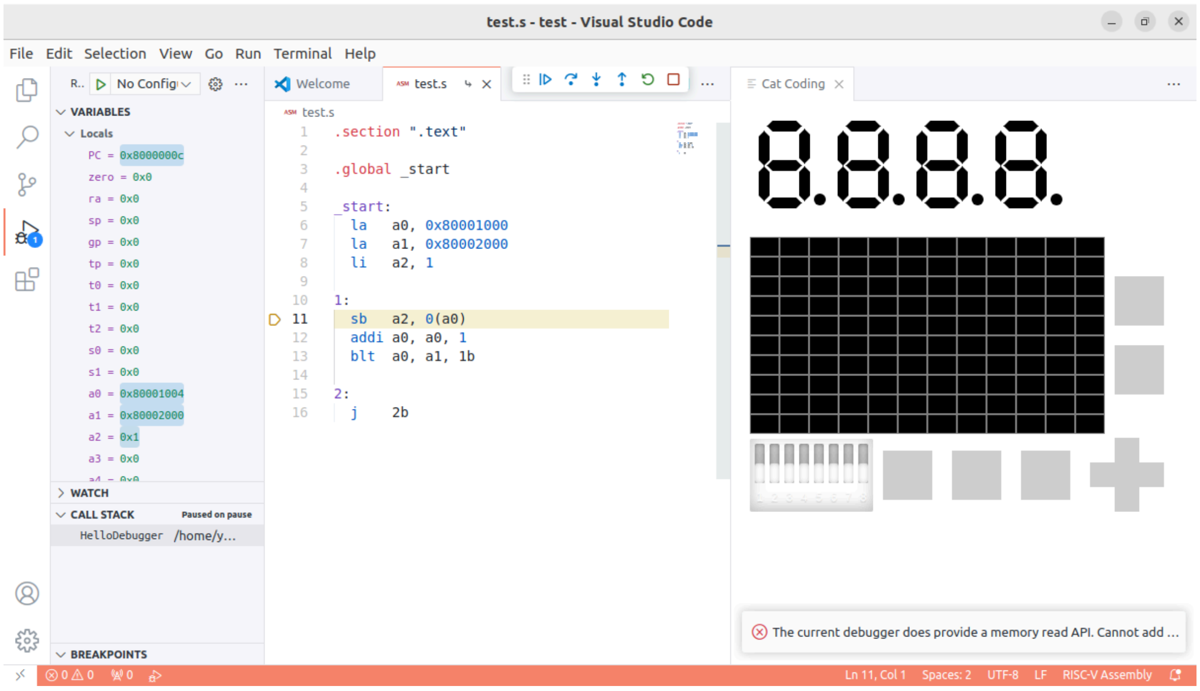
```
PC = 0x80000000
zero = 0x0
ra = 0x0
sp = 0x0
gp = 0x0
tp = 0x0
t0 = 0x0
t1 = 0x0
t2 = 0x0
s0 = 0x0
s1 = 0x0
a0 = 0x0
a1 = 0x0
a2 = 0x0
a3 = 0x0
```

```
HelloDebugger /home/y...
```

```
.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
```

where on the left panel the registers and call stack 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.


test.s - test - Visual Studio Code
File Edit Selection View Go Run Terminal Help
No Config
Welcome
test.s
Cat Coding
VARIABLES
Locals
WATCH
CALL STACK
Paused on pause
BREAKPOINTS
Ln 11, Col 1 Spaces: 2 UTF-8 LF RISC-V Assembly

```
PC = 0x8000000c
zero = 0x0
ra = 0x0
sp = 0x0
gp = 0x0
tp = 0x0
t0 = 0x0
t1 = 0x0
t2 = 0x0
s0 = 0x0
s1 = 0x0
a0 = 0x80001004
a1 = 0x80002000
a2 = 0x1
a3 = 0x0
```

```
HelloDebugger /home/y...
```

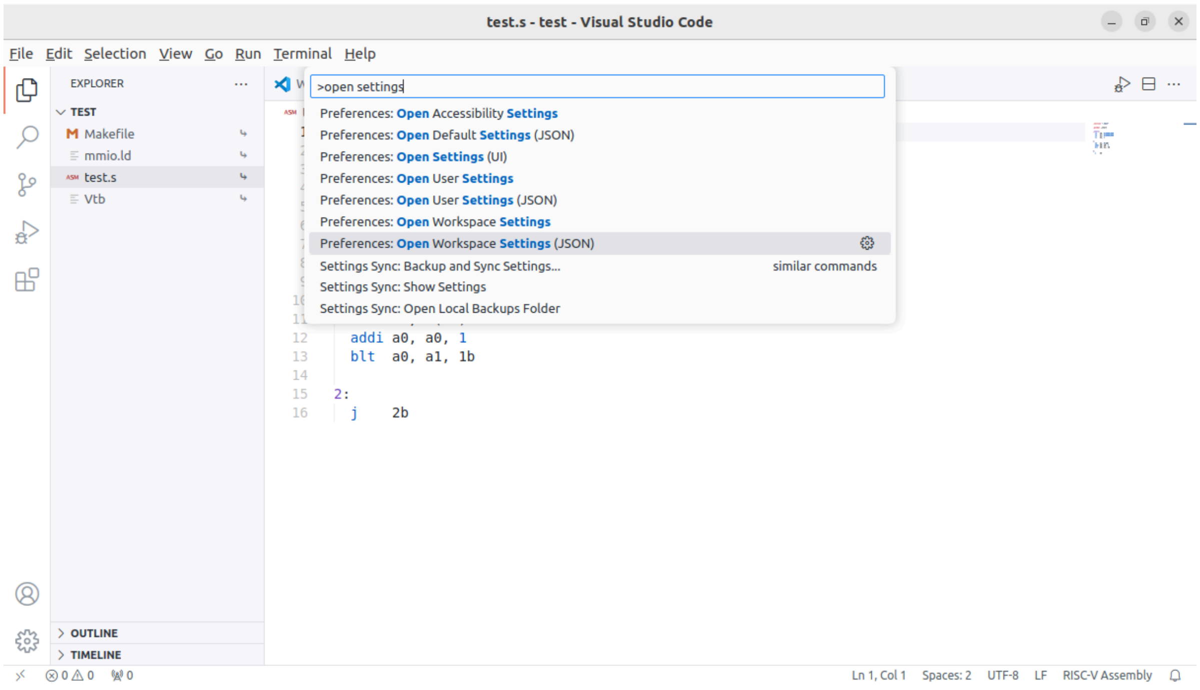
```
.section ".text"

.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
```

```
The current debugger does provide a memory read API. Cannot add ...
```

Setting up memory view

You can also visualize the memory contents and updates with `MemoryView`. You need to go over the following steps to enable it:

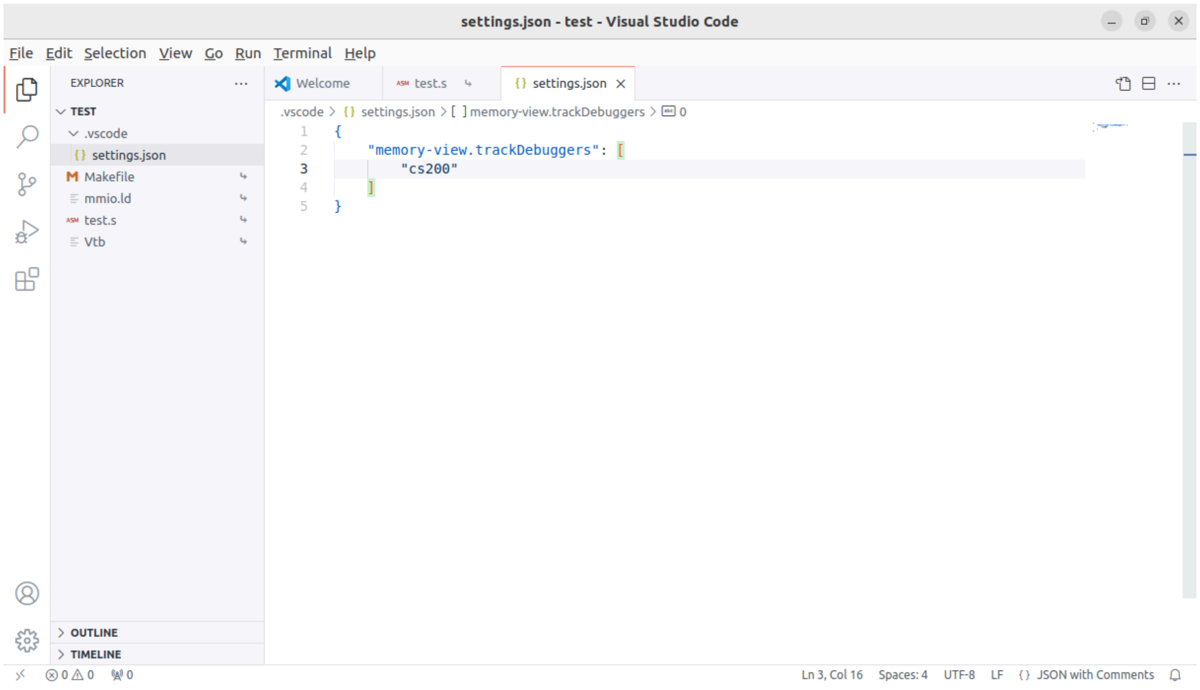
1. Open the command palatte by pressing `Ctrl` + `P`, enter `>open settings`, and select `Preferences: Open Workspace Settings (JSON)`.


test.s - test - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
TEST
Makefile
mmio.ld
test.s
Vtb
OUTLINE
TIMELINE
Ln 1, Col 1 Spaces: 2 UTF-8 LF RISC-V Assembly

```
>open settings
Preferences: Open Accessibility Settings
Preferences: Open Default Settings (JSON)
Preferences: Open Settings (UI)
Preferences: Open User Settings
Preferences: Open User Settings (JSON)
Preferences: Open Workspace Settings
Preferences: Open Workspace Settings (JSON)
Settings Sync: Backup and Sync Settings...          similar commands
Settings Sync: Show Settings
Settings Sync: Open Local Backups Folder
```

```
    addi a0, a0, 1
    blt  a0, a1, 1b

2:
    j    2b
```

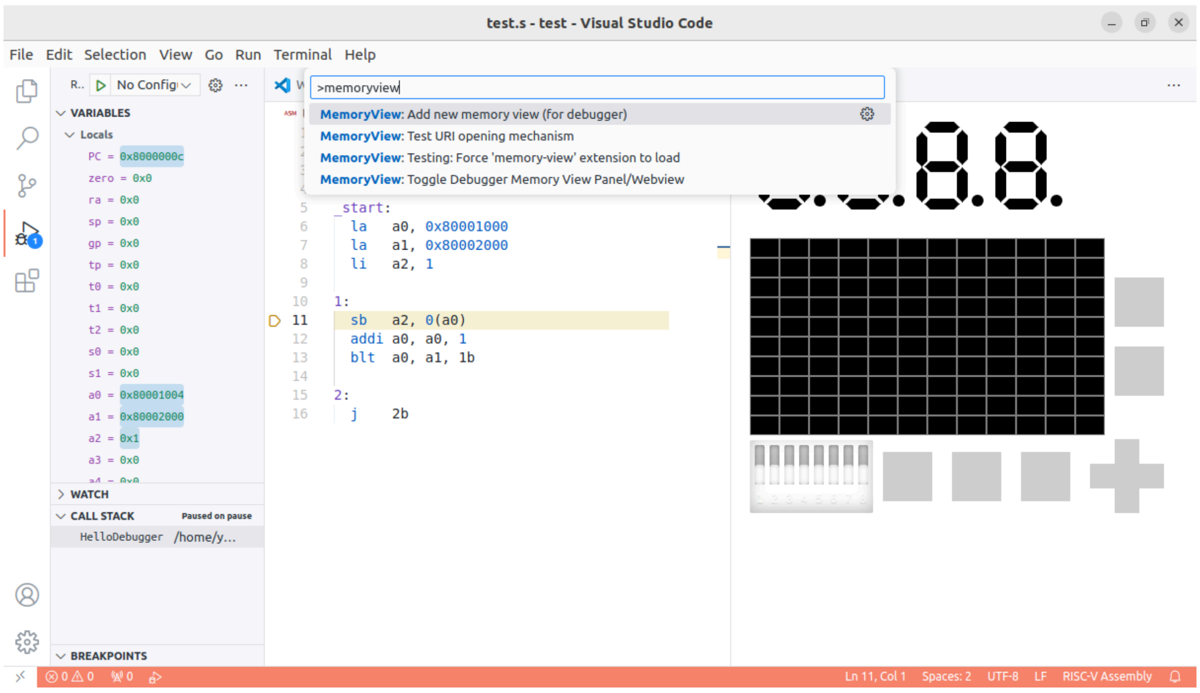
2. In the opened `settings.json` file, add the following content to allow the `MemoryView` extension interacting with the `cs200` debugger:


settings.json - test - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
TEST
.vscode
settings.json
Makefile
mmio.ld
test.s
Vtb
Welcome
test.s
settings.json
.vscode > settings.json > [ ] memory-view.trackDebuggers > 0
OUTLINE
TIMELINE
Ln 3, Col 16 Spaces: 4 UTF-8 LF JSON with Comments

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

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)`.



test.s - test - Visual Studio Code

File Edit Selection View Go Run Terminal Help

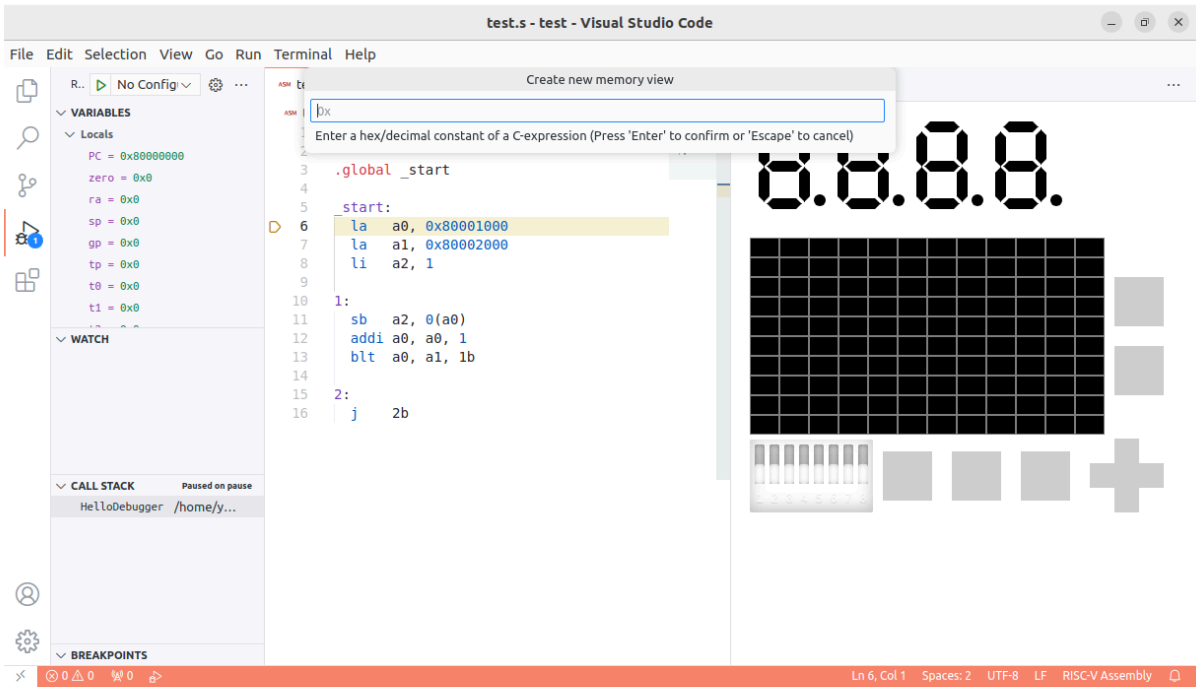
```
>memoryview
MemoryView: Add new memory view (for debugger)
MemoryView: Test URI opening mechanism
MemoryView: Testing: Force 'memory-view' extension to load
MemoryView: Toggle Debugger Memory View Panel/Webview
```

```
VARIABLES
  Locals
    PC = 0x8000000c
    zero = 0x0
    ra = 0x0
    sp = 0x0
    gp = 0x0
    tp = 0x0
    t0 = 0x0
    t1 = 0x0
    t2 = 0x0
    s0 = 0x0
    s1 = 0x0
    a0 = 0x80001004
    a1 = 0x80002000
    a2 = 0x1
    a3 = 0x0
WATCH
CALL STACK            Paused on pause
  HelloDebugger  /home/y...
BREAKPOINTS
```

```
_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
```

Ln 11, Col 1   Spaces: 2   UTF-8   LF   RISC-V Assembly

5. Enter the base address of the memory you want to visualize. In our case, the updated memory starts at `0x80001000`.



test.s - test - Visual Studio Code

File Edit Selection View Go Run Terminal Help

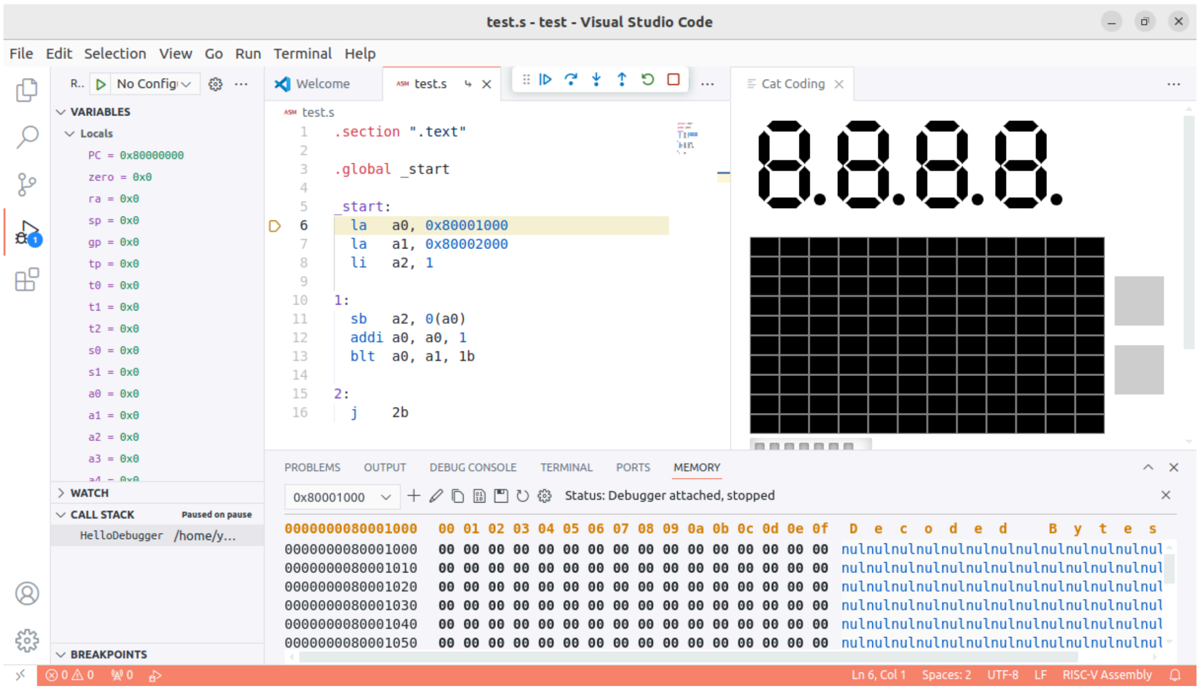
```
Create new memory view
0x
Enter a hex/decimal constant of a C-expression (Press 'Enter' to confirm or 'Escape' to cancel)
```

```
VARIABLES
  Locals
    PC = 0x80000000
    zero = 0x0
    ra = 0x0
    sp = 0x0
    gp = 0x0
    tp = 0x0
    t0 = 0x0
    t1 = 0x0
WATCH
CALL STACK            Paused on pause
  HelloDebugger  /home/y...
BREAKPOINTS
```

```
.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
```

Ln 6, Col 1   Spaces: 2   UTF-8   LF   RISC-V Assembly

6. The memory view will be shown as follows:



test.s - test - Visual Studio Code

```
PC = 0x80000000
zero = 0x0
ra = 0x0
sp = 0x0
gp = 0x0
tp = 0x0
t0 = 0x0
t1 = 0x0
t2 = 0x0
s0 = 0x0
s1 = 0x0
a0 = 0x0
a1 = 0x0
a2 = 0x0
a3 = 0x0
```

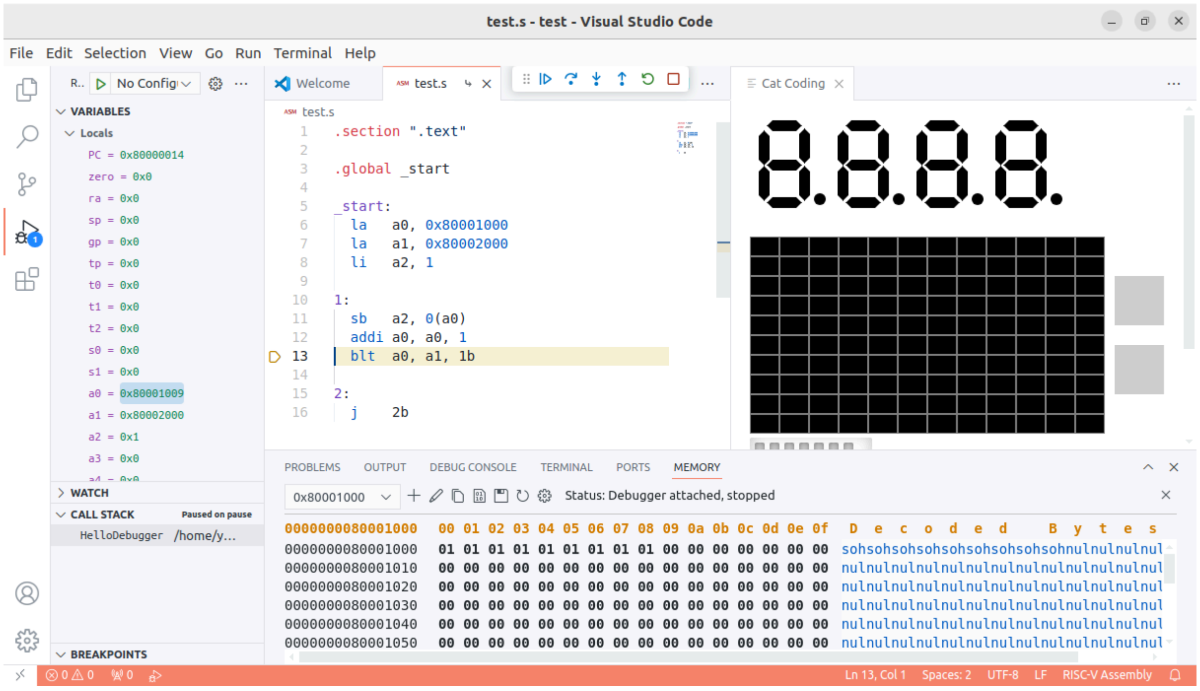
```
.section ".text"

.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
```

```
0x80001000    Status: Debugger attached, stopped

0000000080001000  00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f  D e c o d e d   B y t e s
0000000080001000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  nulnulnulnulnulnulnulnulnulnulnulnulnulnulnulnul
0000000080001010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  nulnulnulnulnulnulnulnulnulnulnulnulnulnulnulnul
0000000080001020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  nulnulnulnulnulnulnulnulnulnulnulnulnulnulnulnul
0000000080001030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  nulnulnulnulnulnulnulnulnulnulnulnulnulnulnulnul
0000000080001040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  nulnulnulnulnulnulnulnulnulnulnulnulnulnulnulnul
0000000080001050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  nulnulnulnulnulnulnulnulnulnulnulnulnulnulnulnul
```

Ln 6, Col 1   Spaces: 2   UTF-8   LF   RISC-V Assembly

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.



test.s - test - Visual Studio Code

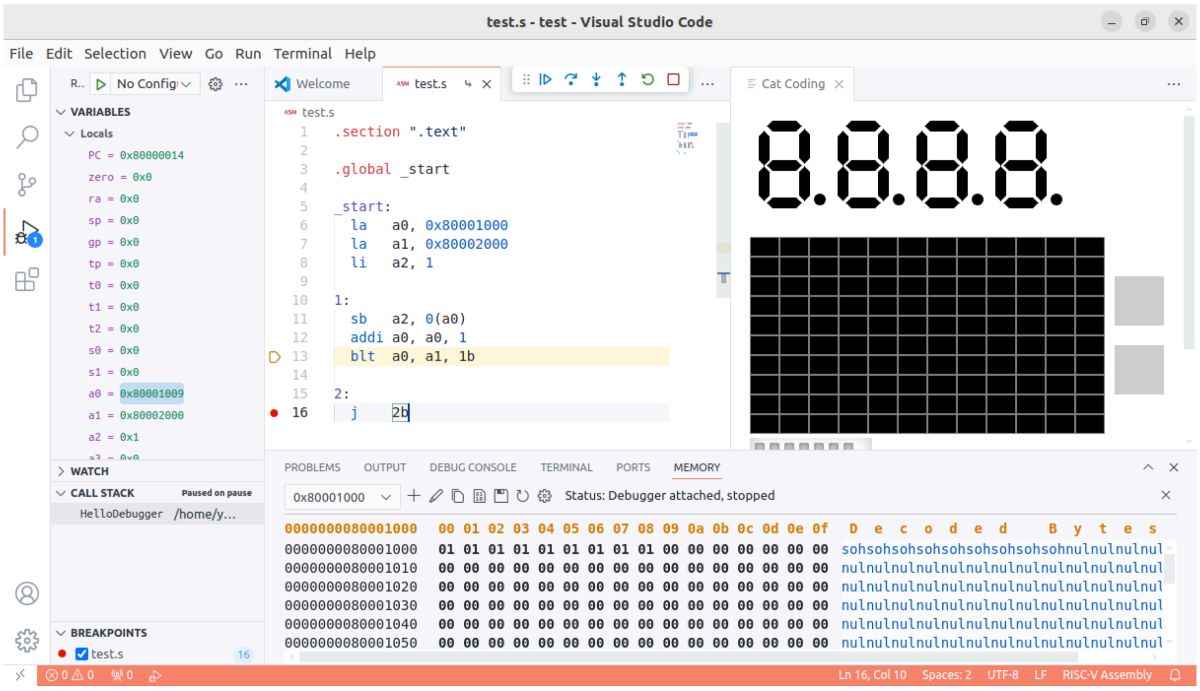
```
PC = 0x80000014
zero = 0x0
ra = 0x0
sp = 0x0
gp = 0x0
tp = 0x0
t0 = 0x0
t1 = 0x0
t2 = 0x0
s0 = 0x0
s1 = 0x0
a0 = 0x80001009
a1 = 0x80002000
a2 = 0x1
a3 = 0x0
```

```
.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
```

```
0x80001000    Status: Debugger attached, stopped

0000000080001000  00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f  D e c o d e d   B y t e s
0000000080001000  01 01 01 01 01 01 01 01 01 00 00 00 00 00 00 00  sohsohsohsohsohsohsohsohsohnulnulnulnulnulnulnul
0000000080001010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  nulnulnulnulnulnulnulnulnulnulnulnulnulnulnulnul
0000000080001020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  nulnulnulnulnulnulnulnulnulnulnulnulnulnulnulnul
0000000080001030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  nulnulnulnulnulnulnulnulnulnulnulnulnulnulnulnul
0000000080001040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  nulnulnulnulnulnulnulnulnulnulnulnulnulnulnulnul
0000000080001050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  nulnulnulnulnulnulnulnulnulnulnulnulnulnulnulnul
```

Ln 13, Col 1   Spaces: 2   UTF-8   LF   RISC-V Assembly

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


test.s - test - Visual Studio Code
File Edit Selection View Go Run Terminal Help
R.. No Config
Welcome
test.s
Cat Coding
VARIABLES
Locals
WATCH
CALL STACK Paused on pause
BREAKPOINTS
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS MEMORY
Ln 16, Col 10 Spaces: 2 UTF-8 LF RISC-V Assembly

```
PC = 0x80000014
zero = 0x0
ra = 0x0
sp = 0x0
gp = 0x0
tp = 0x0
t0 = 0x0
t1 = 0x0
t2 = 0x0
s0 = 0x0
s1 = 0x0
a0 = 0x80001009
a1 = 0x80002000
a2 = 0x1
```

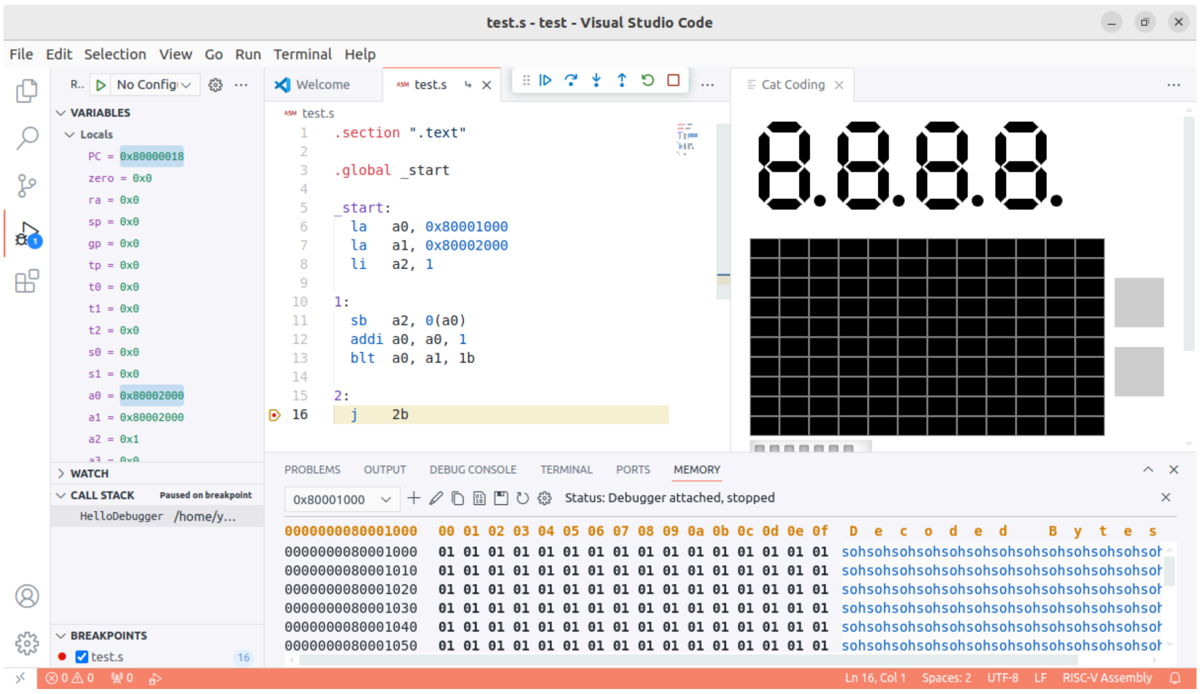
```
HelloDebugger  /home/y...
test.s  16
```

```
.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
```

```
0x80001000   Status: Debugger attached, stopped

0000000080001000  00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f  D e c o d e d   B y t e s
0000000080001000  01 01 01 01 01 01 01 01 01 00 00 00 00 00 00 00  sohsohsohsohsohsohsohsohsohnulnulnulnul
0000000080001010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  nulnulnulnulnulnulnulnulnulnulnulnulnul
0000000080001020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  nulnulnulnulnulnulnulnulnulnulnulnulnul
0000000080001030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  nulnulnulnulnulnulnulnulnulnulnulnulnul
0000000080001040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  nulnulnulnulnulnulnulnulnulnulnulnulnul
0000000080001050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  nulnulnulnulnulnulnulnulnulnulnulnulnul
```

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.

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:


test.s - test - Visual Studio Code
File Edit Selection View Go Run Terminal Help
R.. No Config
Welcome
test.s
Cat Coding
VARIABLES
Locals
WATCH
CALL STACK Paused on breakpoint
BREAKPOINTS
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS MEMORY
Ln 16, Col 1 Spaces: 2 UTF-8 LF RISC-V Assembly

```
PC = 0x80000018
zero = 0x0
ra = 0x0
sp = 0x0
gp = 0x0
tp = 0x0
t0 = 0x0
t1 = 0x0
t2 = 0x0
s0 = 0x0
s1 = 0x0
a0 = 0x80002000
a1 = 0x80002000
a2 = 0x1
```

```
HelloDebugger  /home/y...
test.s  16
```

```
.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
```

```
0x80001000   Status: Debugger attached, stopped

0000000080001000  00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f  D e c o d e d   B y t e s
0000000080001000  01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01  sohsohsohsohsohsohsohsohsohsohsohsohsoh
0000000080001010  01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01  sohsohsohsohsohsohsohsohsohsohsohsohsoh
0000000080001020  01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01  sohsohsohsohsohsohsohsohsohsohsohsohsoh
0000000080001030  01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01  sohsohsohsohsohsohsohsohsohsohsohsohsoh
0000000080001040  01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01  sohsohsohsohsohsohsohsohsohsohsohsohsoh
0000000080001050  01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01  sohsohsohsohsohsohsohsohsohsohsohsohsoh
```

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.

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

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. 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

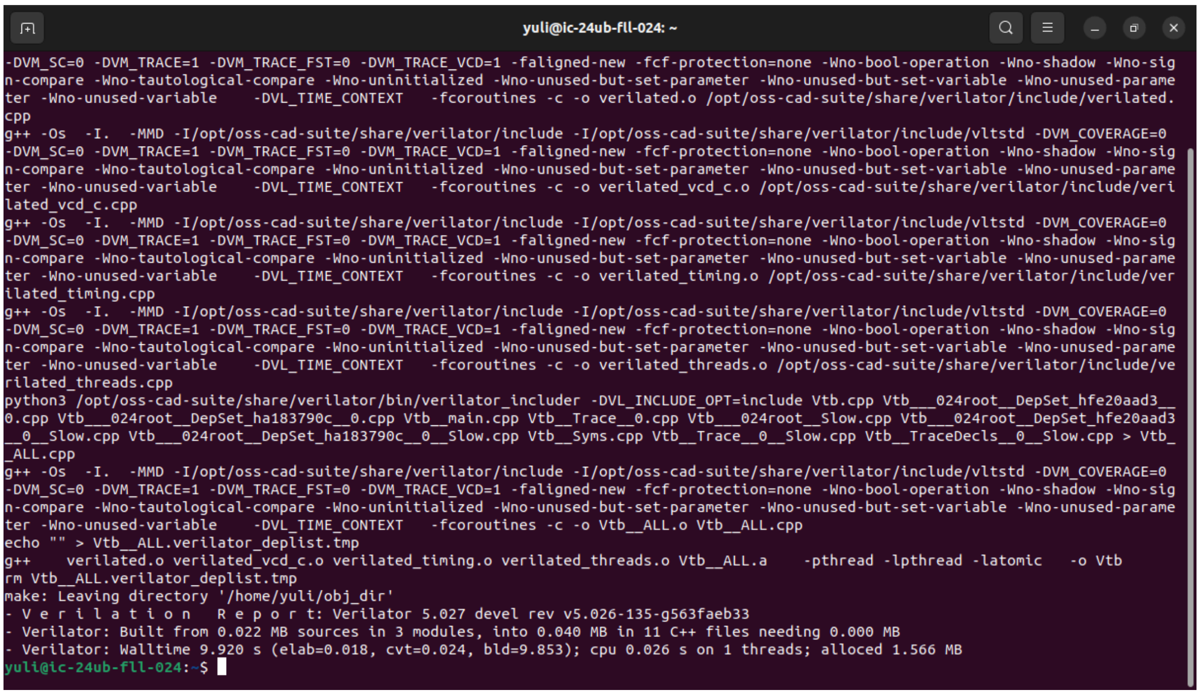
We use verilator to compile the Verilog code into a C++ model, which can later be compiled to an executable to simulate the design.

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
```

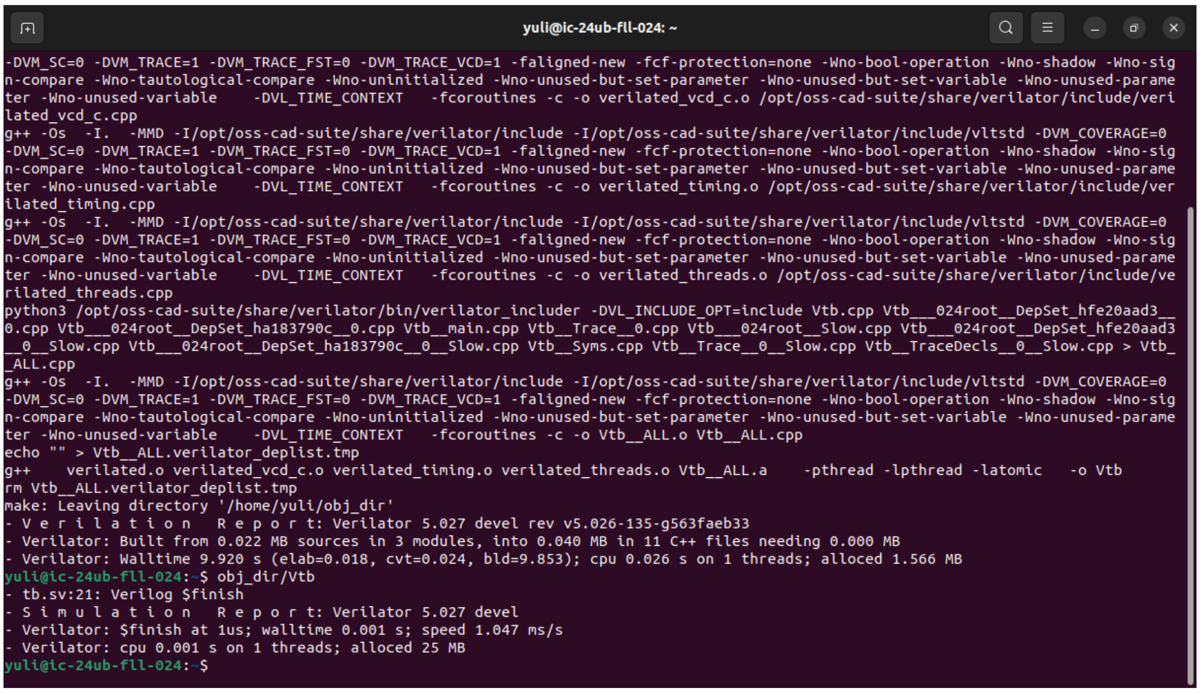
Then you can invoke verilator with the following arguments to compile Verilog into C++ and then build an executable:

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



```
-DVM_SC=0 -DVM_TRACE=1 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=1 -faligned-new -fcf-protection=none -Wno-bool-operation -Wno-shadow -Wno-sig
n-compare -Wno-tautological-compare -Wno-uninitialized -Wno-unused-but-set-parameter -Wno-unused-but-set-variable -Wno-unused-parame
ter -Wno-unused-variable    -DVL_TIME_CONTEXT   -fcoroutines -c -o verilated.o /opt/oss-cad-suite/share/verilator/include/verilated.
cpp
g++ -Os  -I.  -MMD -I/opt/oss-cad-suite/share/verilator/include -I/opt/oss-cad-suite/share/verilator/include/vltstd -DVM_COVERAGE=0
-DVM_SC=0 -DVM_TRACE=1 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=1 -faligned-new -fcf-protection=none -Wno-bool-operation -Wno-shadow -Wno-sig
n-compare -Wno-tautological-compare -Wno-uninitialized -Wno-unused-but-set-parameter -Wno-unused-but-set-variable -Wno-unused-parame
ter -Wno-unused-variable    -DVL_TIME_CONTEXT   -fcoroutines -c -o verilated_vcd_c.o /opt/oss-cad-suite/share/verilator/include/veri
lated_vcd_c.cpp
g++ -Os  -I.  -MMD -I/opt/oss-cad-suite/share/verilator/include -I/opt/oss-cad-suite/share/verilator/include/vltstd -DVM_COVERAGE=0
-DVM_SC=0 -DVM_TRACE=1 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=1 -faligned-new -fcf-protection=none -Wno-bool-operation -Wno-shadow -Wno-sig
n-compare -Wno-tautological-compare -Wno-uninitialized -Wno-unused-but-set-parameter -Wno-unused-but-set-variable -Wno-unused-parame
ter -Wno-unused-variable    -DVL_TIME_CONTEXT   -fcoroutines -c -o verilated_timing.o /opt/oss-cad-suite/share/verilator/include/ver
ilated_timing.cpp
g++ -Os  -I.  -MMD -I/opt/oss-cad-suite/share/verilator/include -I/opt/oss-cad-suite/share/verilator/include/vltstd -DVM_COVERAGE=0
-DVM_SC=0 -DVM_TRACE=1 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=1 -faligned-new -fcf-protection=none -Wno-bool-operation -Wno-shadow -Wno-sig
n-compare -Wno-tautological-compare -Wno-uninitialized -Wno-unused-but-set-parameter -Wno-unused-but-set-variable -Wno-unused-parame
ter -Wno-unused-variable    -DVL_TIME_CONTEXT   -fcoroutines -c -o verilated_threads.o /opt/oss-cad-suite/share/verilator/include/ve
rilated_threads.cpp
python3 /opt/oss-cad-suite/share/verilator/bin/verilator_includer -DVL_INCLUDE_OPT=include Vtb.cpp Vtb___024root__DepSet_hfe20aad3__
0.cpp Vtb___024root__DepSet_ha183790c__0.cpp Vtb__main.cpp Vtb__Trace__0.cpp Vtb___024root__Slow.cpp Vtb___024root__DepSet_hfe20aad3
__0__Slow.cpp Vtb___024root__DepSet_ha183790c__0__Slow.cpp Vtb__Syms.cpp Vtb__Trace__0__Slow.cpp Vtb__TraceDecls__0__Slow.cpp > Vtb_
_ALL.cpp
g++ -Os  -I.  -MMD -I/opt/oss-cad-suite/share/verilator/include -I/opt/oss-cad-suite/share/verilator/include/vltstd -DVM_COVERAGE=0
-DVM_SC=0 -DVM_TRACE=1 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=1 -faligned-new -fcf-protection=none -Wno-bool-operation -Wno-shadow -Wno-sig
n-compare -Wno-tautological-compare -Wno-uninitialized -Wno-unused-but-set-parameter -Wno-unused-but-set-variable -Wno-unused-parame
ter -Wno-unused-variable    -DVL_TIME_CONTEXT   -fcoroutines -c -o Vtb__ALL.o Vtb__ALL.cpp
echo "" > Vtb__ALL.verilator_deplist.tmp
g++    verilated.o verilated_vcd_c.o verilated_timing.o verilated_threads.o Vtb__ALL.a    -pthread -lpthread -latomic   -o Vtb
rm Vtb__ALL.verilator_deplist.tmp
make: Leaving directory '/home/yuli/obj_dir'
- V e r i l a t i o n   R e p o r t: Verilator 5.027 devel rev v5.026-135-g563faeb33
- Verilator: Built from 0.022 MB sources in 3 modules, into 0.040 MB in 11 C++ files needing 0.000 MB
- Verilator: Walltime 9.920 s (elab=0.018, cvt=0.024, bld=9.853); cpu 0.026 s on 1 threads; alloced 1.566 MB
yuli@ic-24ub-fll-024:~$
```

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



```
yuli@ic-24ub-fll-024: ~
-DVM_SC=0 -DVM_TRACE=1 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=1 -faligned-new -fcf-protection=none -Wno-bool-operation -Wno-shadow -Wno-sig
n-compare -Wno-tautological-compare -Wno-uninitialized -Wno-unused-but-set-parameter -Wno-unused-but-set-variable -Wno-unused-parame
ter -Wno-unused-variable    -DVL_TIME_CONTEXT   -fcoroutines -c -o verilated_vcd_c.o /opt/oss-cad-suite/share/verilator/include/veri
lated_vcd_c.cpp
g++ -Os  -I.  -MMD -I/opt/oss-cad-suite/share/verilator/include -I/opt/oss-cad-suite/share/verilator/include/vltstd -DVM_COVERAGE=0
-DVM_SC=0 -DVM_TRACE=1 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=1 -faligned-new -fcf-protection=none -Wno-bool-operation -Wno-shadow -Wno-sig
n-compare -Wno-tautological-compare -Wno-uninitialized -Wno-unused-but-set-parameter -Wno-unused-but-set-variable -Wno-unused-parame
ter -Wno-unused-variable    -DVL_TIME_CONTEXT   -fcoroutines -c -o verilated_timing.o /opt/oss-cad-suite/share/verilator/include/ver
ilated_timing.cpp
g++ -Os  -I.  -MMD -I/opt/oss-cad-suite/share/verilator/include -I/opt/oss-cad-suite/share/verilator/include/vltstd -DVM_COVERAGE=0
-DVM_SC=0 -DVM_TRACE=1 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=1 -faligned-new -fcf-protection=none -Wno-bool-operation -Wno-shadow -Wno-sig
n-compare -Wno-tautological-compare -Wno-uninitialized -Wno-unused-but-set-parameter -Wno-unused-but-set-variable -Wno-unused-parame
ter -Wno-unused-variable    -DVL_TIME_CONTEXT   -fcoroutines -c -o verilated_threads.o /opt/oss-cad-suite/share/verilator/include/ve
rilated_threads.cpp
python3 /opt/oss-cad-suite/share/verilator/bin/verilator_includer -DVL_INCLUDE_OPT=include Vtb.cpp Vtb___024root__DepSet_hfe20aad3__
0.cpp Vtb___024root__DepSet_ha183790c__0.cpp Vtb__main.cpp Vtb__Trace__0.cpp Vtb___024root__Slow.cpp Vtb___024root__DepSet_hfe20aad3
__0__Slow.cpp Vtb___024root__DepSet_ha183790c__0__Slow.cpp Vtb__Syms.cpp Vtb__Trace__0__Slow.cpp Vtb__TraceDecls__0__Slow.cpp > Vtb_
_ALL.cpp
g++ -Os  -I.  -MMD -I/opt/oss-cad-suite/share/verilator/include -I/opt/oss-cad-suite/share/verilator/include/vltstd -DVM_COVERAGE=0
-DVM_SC=0 -DVM_TRACE=1 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=1 -faligned-new -fcf-protection=none -Wno-bool-operation -Wno-shadow -Wno-sig
n-compare -Wno-tautological-compare -Wno-uninitialized -Wno-unused-but-set-parameter -Wno-unused-but-set-variable -Wno-unused-parame
ter -Wno-unused-variable    -DVL_TIME_CONTEXT   -fcoroutines -c -o Vtb__ALL.o Vtb__ALL.cpp
echo "" > Vtb__ALL.verilator_deplist.tmp
g++    verilated.o verilated_vcd_c.o verilated_timing.o verilated_threads.o Vtb__ALL.a    -pthread -lpthread -latomic   -o Vtb
rm Vtb__ALL.verilator_deplist.tmp
make: Leaving directory '/home/yuli/obj_dir'
- V e r i l a t i o n   R e p o r t: Verilator 5.027 devel rev v5.026-135-g563faeb33
- Verilator: Built from 0.022 MB sources in 3 modules, into 0.040 MB in 11 C++ files needing 0.000 MB
- Verilator: Walltime 9.920 s (elab=0.018, cvt=0.024, bld=9.853); cpu 0.026 s on 1 threads; alloced 1.566 MB
yuli@ic-24ub-fll-024:~$ obj_dir/Vtb
- tb.sv:21: Verilog $finish
- S i m u l a t i o n   R e p o r t: Verilator 5.027 devel
- Verilator: $finish at 1us; walltime 0.001 s; speed 1.047 ms/s
- Verilator: cpu 0.001 s on 1 threads; alloced 25 MB
yuli@ic-24ub-fll-024:~$
```

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

Using GTKWave

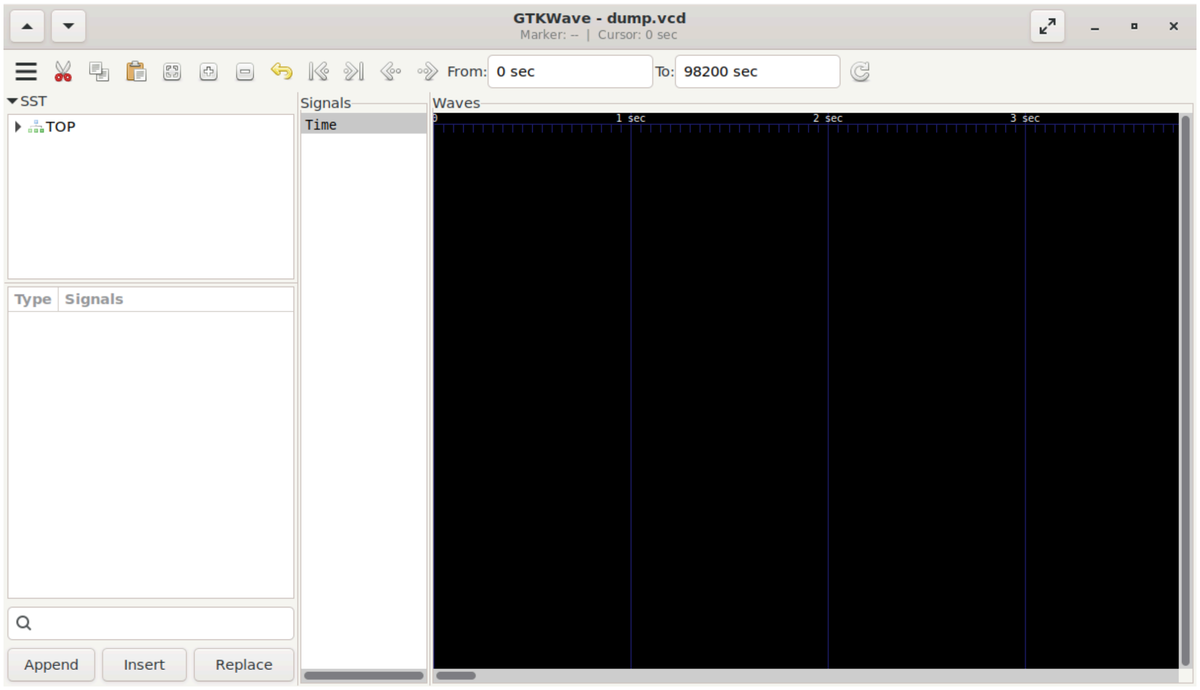
The compiled model, when executing, can generate a waveform file that contains the values of all the signals in the design at each time step. We use GTKWave to visualize the waveform and help us debug the design.

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

```
gtkwave dump.vcd
```

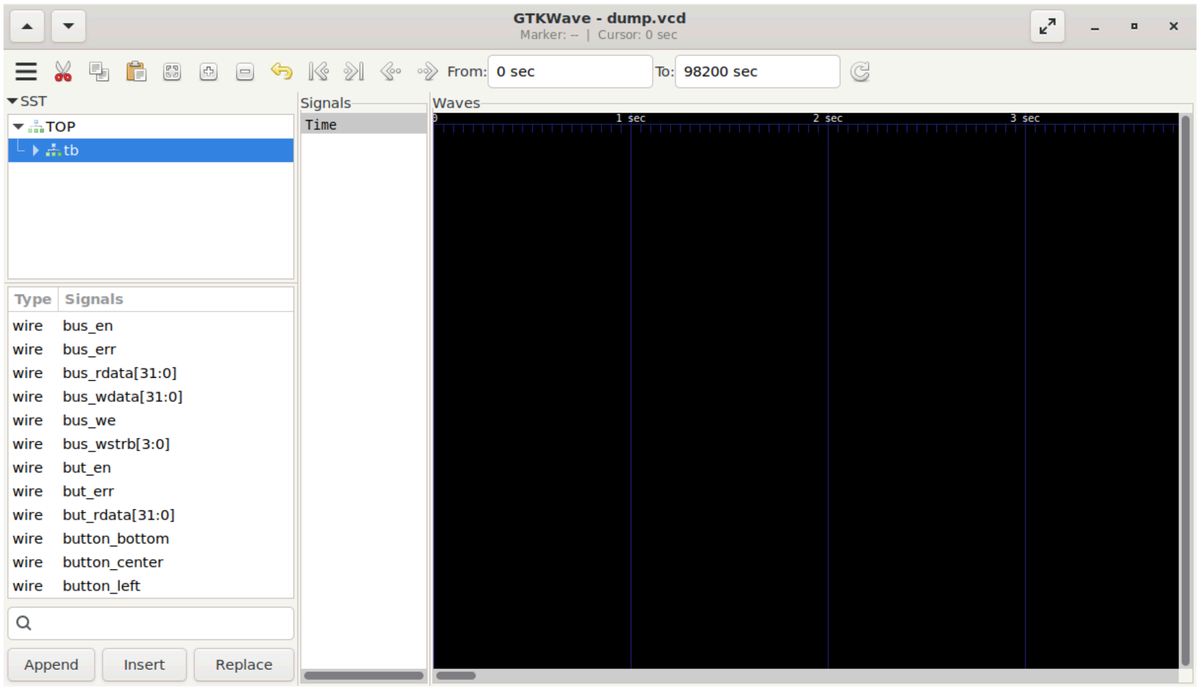
where `dump.vcd` is the waveform file generated by the simulation.

The GTKWave GUI looks as follows:



GTKWave - dump.vcd
Marker: -- | Cursor: 0 sec
From: 0 sec To: 98200 sec

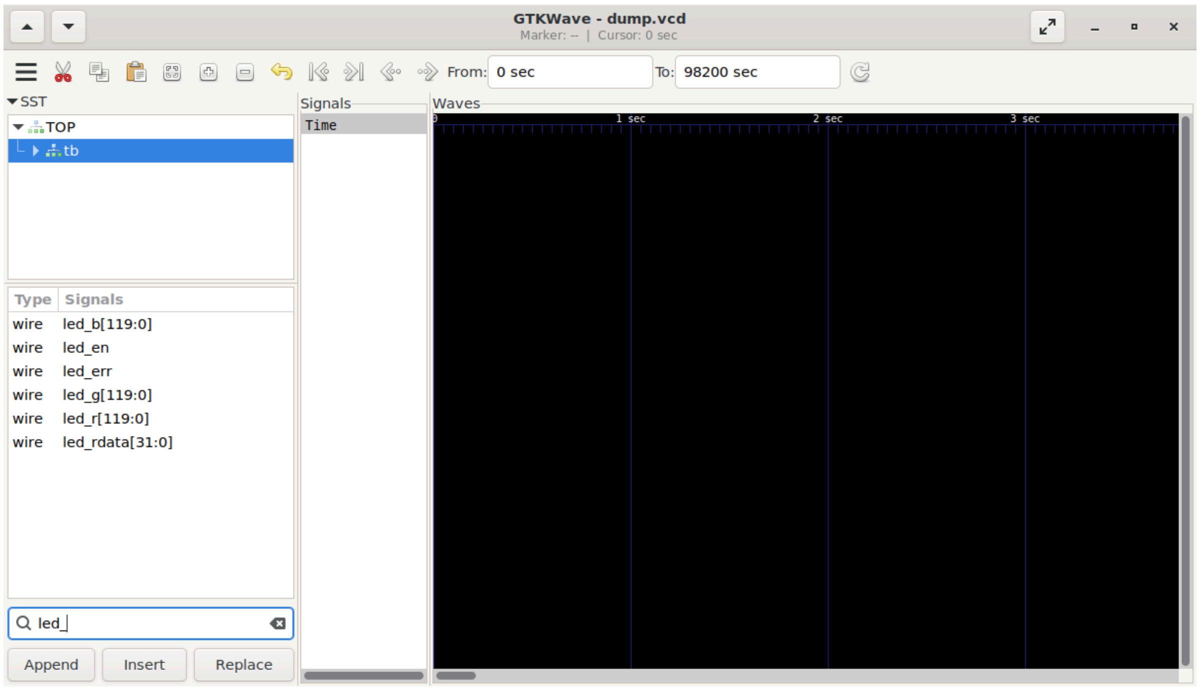
In the left panel, you can see the signal search tree (SST) that shows the hierarchy of the design. You can expand each module to see all its submodules. When a module is selected, all its signals will be shown in the below list:


GTKWave - dump.vcd
Marker: -- | Cursor: 0 sec
From: 0 sec To: 98200 sec

SST
- TOP
  - tb

Type	Signals
wire	bus_en
wire	bus_err
wire	bus_rdata[31:0]
wire	bus_wdata[31:0]
wire	bus_we
wire	bus_wstrb[3:0]
wire	but_en
wire	but_err
wire	but_rdata[31:0]
wire	button_bottom
wire	button_center
wire	button_left

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:

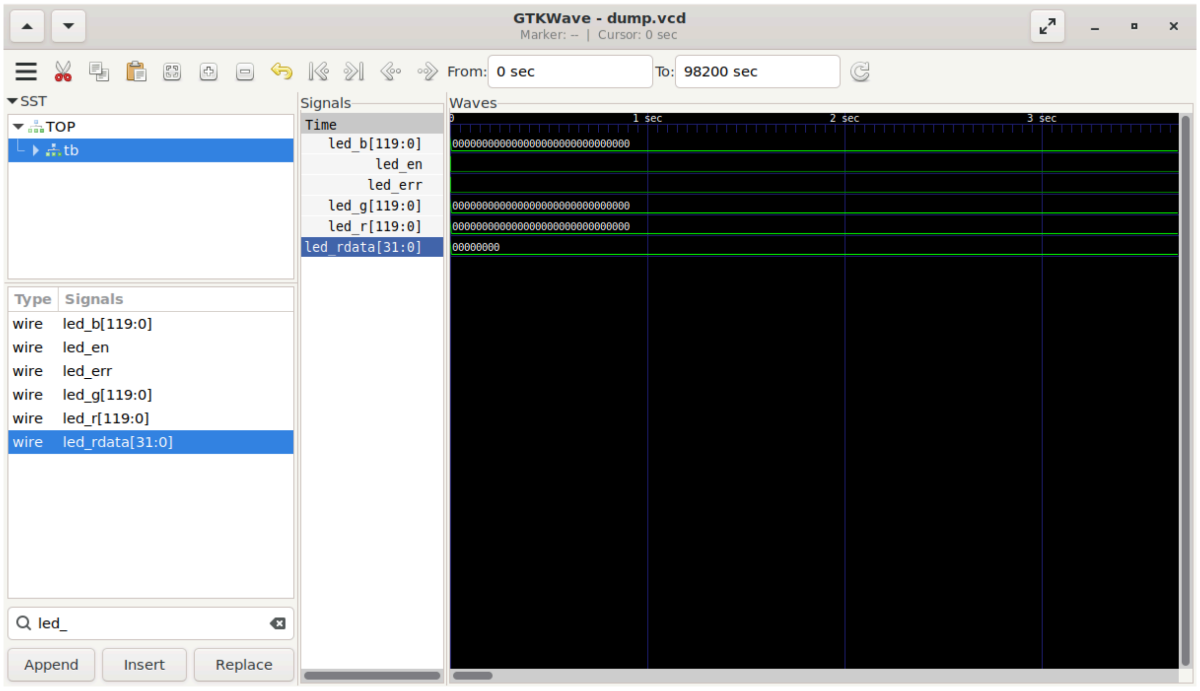

GTKWave - dump.vcd
Marker: -- | Cursor: 0 sec
From: 0 sec To: 98200 sec

SST
- TOP
  - tb

Type	Signals
wire	led_b[119:0]
wire	led_en
wire	led_err
wire	led_g[119:0]
wire	led_r[119:0]
wire	led_rdata[31:0]

led_

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:



From: 0 sec To: 98200 sec

Type	Signals
wire	led_b[119:0]
wire	led_en
wire	led_err
wire	led_g[119:0]
wire	led_r[119:0]
wire	led_rdata[31:0]

led_

```
led_b[119:0]     000000000000000000000000000000
led_en
led_err
led_g[119:0]     000000000000000000000000000000
led_r[119:0]     000000000000000000000000000000
led_rdata[31:0]  00000000
```

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.