Virtual machines

As part of CS-173, you will be expected to simulate and analyse circuits, such as those of a CPU.

All required software is preinstalled on INF 3 computers and the virtual desktop infrastructure (VDI) virtual machines (VMs) that you can access remotely.

To ensure that all students have access to the required software, and to minimise platform- and user-specific issues, you are expected and encouraged to work in INF 3 or VDI. You may of course also work on your own machines, but there is no guarantee that the teaching staff will be able to help you resolve issues specific to your setup.

Accessing physical machines

1. Make your way to computer lab INF 3.
2. Log into any available desktop using your GASPAR credentials.
3. That's it!

Accessing VDI remotely

EPFL's VDI uses the VMware Horizon service, which can be accessed via either the browser or a locally installed client.

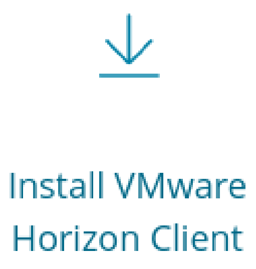
Note that the browser version is more limited than the VMware Horizon Client.

VMware Horizon Client

1. Navigate to https://vdi.epfl.ch.

2. Select and run the most appropriate installer for your system:

    - either click on `Install VMware Horizon Client` to download the default installer:



    - or click on `full list of VMware Horizon Clients` to view alternative options:

To see the full list of VMware Horizon Clients, click here.

3. Launch VMware Horizon Client (also named `vmware-view` on some platforms).
4. Add https://vdi.epfl.ch as a New Server.
5. Connect to it using your GASPAR credentials.
6. Select the `IC-CO-IN-SC-INJ-2025-Spring` machine.
7. You should now be faced with the same Ubuntu desktop as you would see in INF 3.

Browser

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

3. Log in with your GASPAR credentials.
4. Click on the `IC-CO-IN-SC-INJ-2025-Spring` machine.
5. You should now be faced with the same Ubuntu desktop as you would see in INF 3.

Persisting files across sessions and machines

Under the hood, the physical computers in INF 3 are actually connecting to the same VMs as you would via VDI. This means that your files and sessions can be shared, to an extent, across logins and machines.

N.B.: As the default VDI wallpaper warns, not all files are persisted across sessions or reboots. To **avoid losing data**, always place important files under the `~/Desktop/myfiles/` directory!

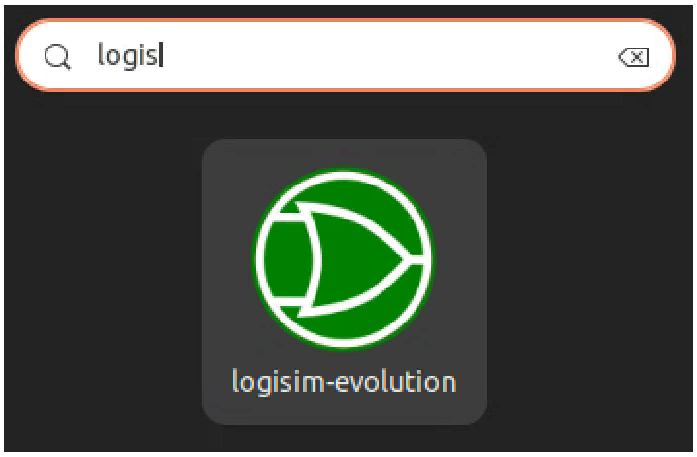
Logisim-evolution

During the first part of the course, we will start simulating simple circuits in Logisim-evolution.

You can open it by either launching the `/opt/logisim-evolution/bin/logisim-evolution` executable, or via the following steps:

1. Open the menu by pressing the `Super/Windows/Opt` key on your keyboard, or by clicking on Ubuntu's `Show Applications` button in the bottom-left corner:

2. Search for and select the `logisim-evolution` application:



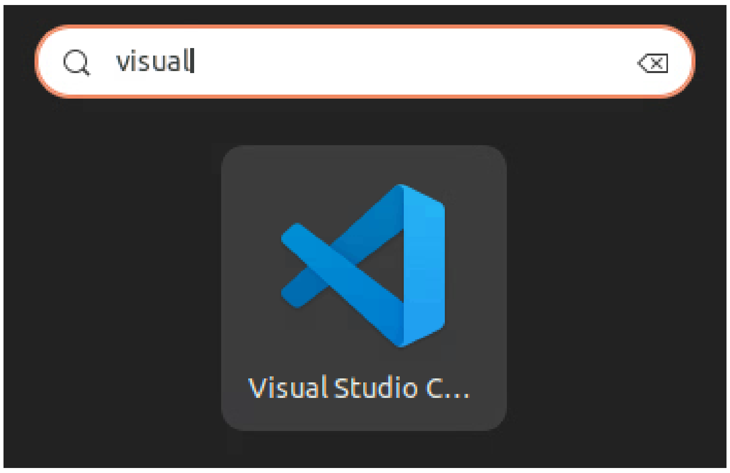
N.B.: Remember to **save your work** under the `~/Desktop/myfiles/` directory **often**, to avoid losing your data if the network disconnects or something else goes wrong!

VSCode

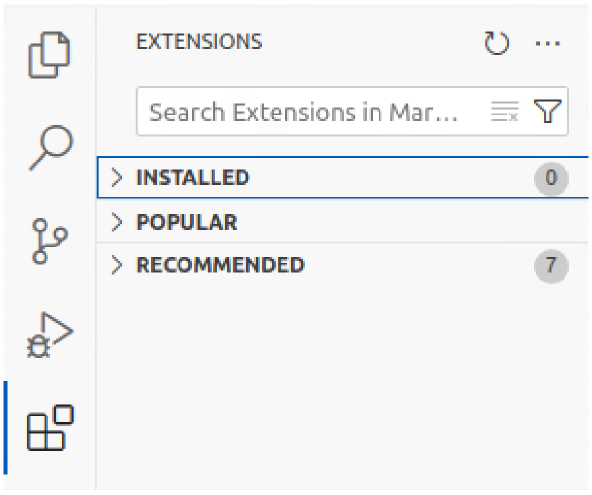
You are most welcome to use your preferred editor or development environment, but the standard setup for this course will involve the use of the Visual Studio Code (VSCode) IDE.

Here is how you can set it up on VDI:

1. Open Visual Studio Code by either launching the `code` executable, or via the same steps as above, replacing `logisim-evolution` with `Visual Studio Code`:



2. Click on `Extensions` in the left sidebar (or type `Ctrl+Shift+x`):



3. Search for and install the following extensions:
   1. Markdown All in One by Yu Zhang

      `yzhang.markdown-all-in-one`
   2. Verilog-HDL/SystemVerilog/Bluespec SystemVerilog by Masahiro Hiramori

      `mshr-h.VerilogHDL`
   3. RISC-V Support by zhwu95

      `zhwu95.riscv`
   4. RISC-V Venus Simulator by hm

      `hm.riscv-venus`

Icarus Verilog

Designing logic circuits manually, as we have done in Logisim-evolution, ceases to be practical when dealing with large, real-world circuits. Instead, we use hardware description languages (HDLs) to describe circuits and how they compose at a high level. From this textual description, a compiler can generate the corresponding schematic, and an interpreter can simulate and test the circuit's behaviour.

In this course, we will write HDL descriptions in Verilog, and use the Icarus Verilog compiler for simulation.

Toy example

Verilog files are usually given the `.v` file extension. Their contents define a `module` (usually named like the file), its inputs and outputs, and its internal behaviour. For example:

```verilog
// hello.v
module hello();

initial begin
  $display("Hello, World!");
  $finish;
end

endmodule
```

This defines a module called `hello` with an empty list `()` of inputs and outputs. All the module does is display the specified greeting.

VSCode setup

If you are using the aforementioned Verilog extension, you might want to configure it to use Icarus Verilog by default:

1. Open VSCode settings (*File* > *Preferences* > *Settings* or `Ctrl+,`).
2. Search for `verilog linter`.
3. Set *Verilog* > *Linting: Linter* to `iverilog`.

The command line examples that follow can all be run either within VSCode's terminal emulator (*View* > *Terminal* or `` Ctrl+` ``), or in a separate terminal program.

CLI

Icarus Verilog is invoked via the `iverilog` executable. Calling it with no arguments results in a brief synopsis of the command-line options it accepts:

```
$ iverilog
/opt/oss-cad-suite/libexec/iverilog: no source files.

Usage: iverilog [-EiRSuvV] [-B base] [-c cmdfile|-f cmdfile]
                [-g1995|-g2001|-g2005|-g2005-sv|-g2009|-g2012]
                [-g<feature>]
                [-D macro[=defn]] [-I includedir] [-L moduledir]
                [-M [mode=]depfile] [-m module]
                [-N file] [-o filename] [-p flag=value]
                [-s topmodule] [-t target] [-T min|typ|max]
                [-W class] [-y dir] [-Y suf] [-l file]
                source_file(s)

See the man page for details.
```

Its `man` (manual) page goes into further details, and can be found at:

```
$ man -l /opt/oss-cad-suite/share/man/man1/iverilog.1
```

Compilation

To compile the toy example above, simply pass its file name to `iverilog`:

```
$ iverilog hello.v
```

By default, this generates an executable file called `a.out`:

```
$ ./a.out
Hello, World!
hello.v:5: $finish called at 0 (1s)
```

You can specify an alternative output file name with the `-o` option:

```
$ iverilog -o hello hello.v
$ ./hello
Hello, World!
hello.v:5: $finish called at 0 (1s)
```

GTKWave

Icarus Verilog is able to generate waveform simulation models from circuit descriptions, but we will use a separate tool for visualising and debugging simulation results: the GTKWave analyser.

To demonstrate how it works, let's use a slightly more complicated Verilog example, namely a 3-input XOR gate:

```
// my_xor.v
module my_xor (
  // One output signal.
  output f,
  // Three input signals.
  input a, b, c
);

// Assign f to be the xor of a, b, and c.
xor(f, a, b, c);

endmodule
```

This describes the circuit's functionality, but it does not provide any test inputs for simulating it. We can script a simple simulation in a separate file as follows (the `tb` in the file name stands for *test bench*):

```
// my_xor_tb.v
module test_my_xor;

// Define three inputs that can store a value.
reg a, b, c;
// And one output that responds to them.
wire f;

// Connect them to an instance of my_xor that we name my_gate.
my_xor my_gate(f, a, b, c);

initial begin
  // Write this test's data to a .vcd file that GTKWave can read.
  $dumpfile("my_xor.vcd");
  $dumpvars(0, test_my_xor);

  // Print values whenever they change.
  $monitor("At time %2t, a=%b b=%b c=%b f=%b", $time, a, b, c, f);

  // Start with all inputs zero.
  a = 0; b = 0; c = 0;
  // Create a small time delay of 5 time units.
  #5

  // Toggle each input with small delays in-between.
  a = 1; #5 b = 1; #5 c = 1; #5

  // Done.
  $finish;
end

endmodule
```

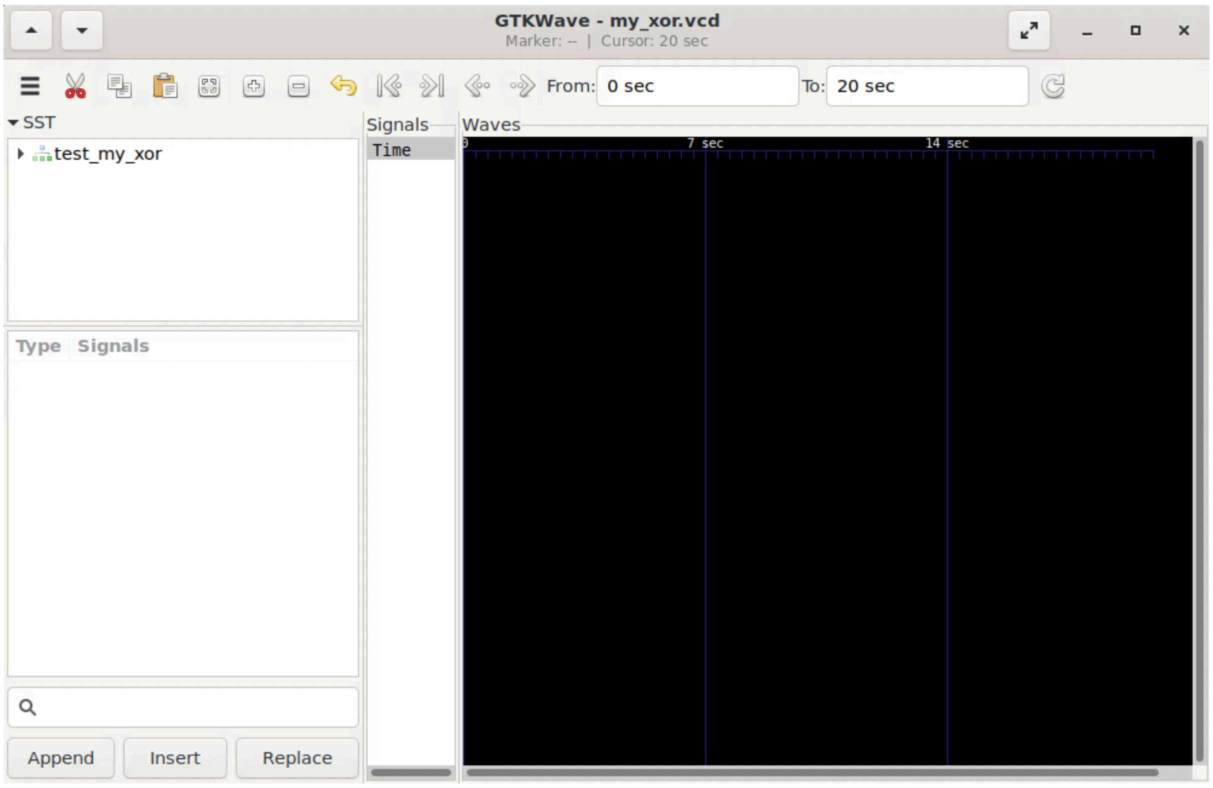
Compile and run this as before, but specify both file names on the command line:

```
$ iverilog -o my_xor my_xor.v my_xor_tb.v
$ ./my_xor
VCD info: dumpfile my_xor.vcd opened for output.
At time  0, a=0 b=0 c=0 f=0
At time  5, a=1 b=0 c=0 f=1
At time 10, a=1 b=1 c=0 f=0
At time 15, a=1 b=1 c=1 f=1
my_xor_tb.v:28: $finish called at 20 (1s)
```

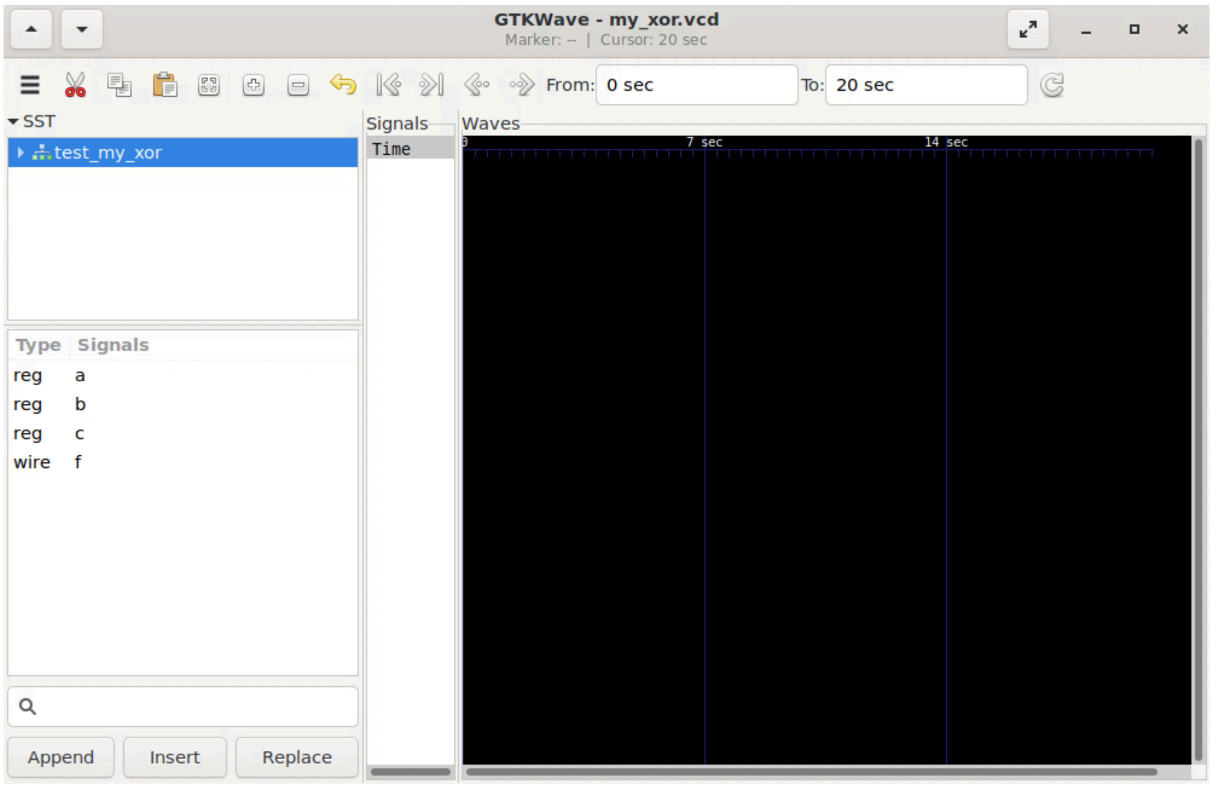
Looks good, but it would look even better as a waveform. First, launch GTKWave with the generated .vcd file:

```
$ gtkwave my_xor.vcd
```

The initial display is empty until we specify which signals we are interested in:

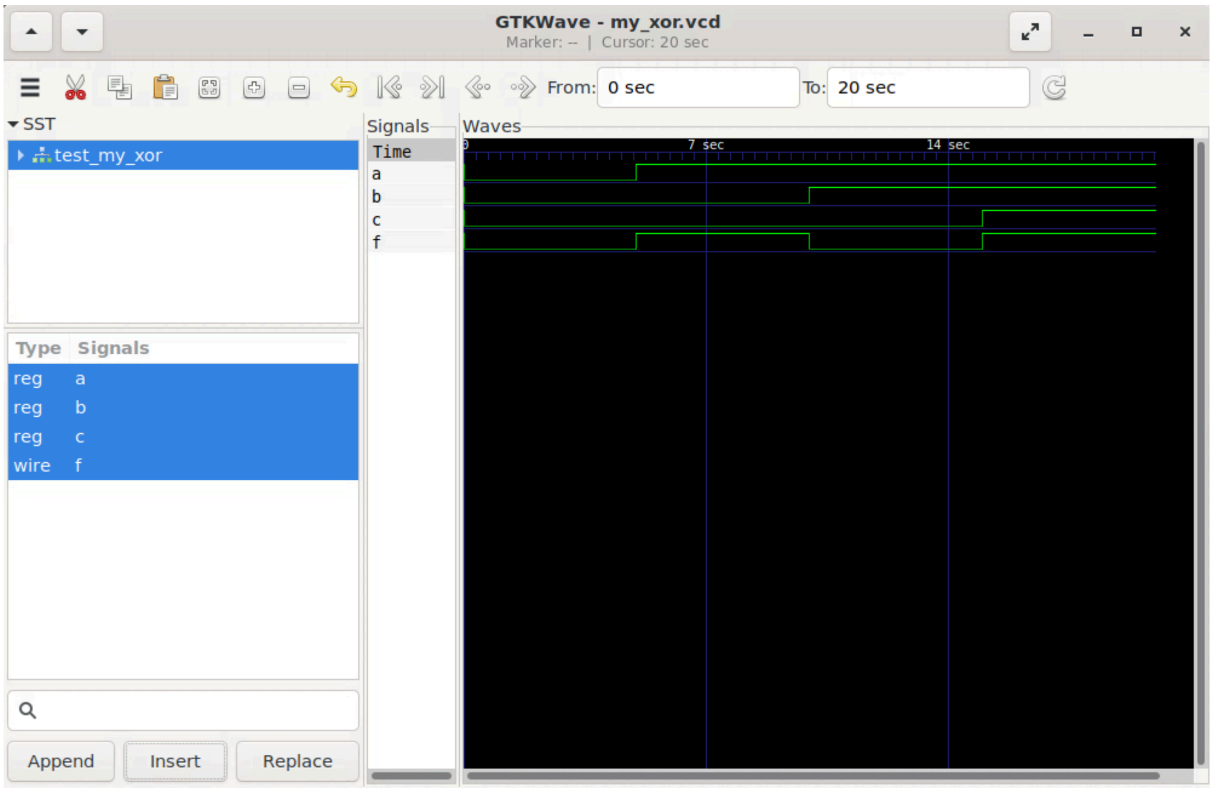


In the top-left SST pane, click on `test_my_xor`. A table of signals should appear below that:



Type	Signals
reg	a
reg	b
reg	c
wire	f

Select them all and either click on the `Insert` button at the bottom, or drag them with the mouse to the `Signals` pane in the middle. You should now be able to see a timing diagram for our 3-input XOR gate:



RISC-V

During the final part of the course, we will start looking at and writing simple assembly programs for the RISC-V ISA.

Simulation

One of the simplest ways to get started with writing and running RISC-V programs is via the two aforementioned VSCode extensions: RISC-V Support, which adds syntax highlighting and snippets, and RISC-V Venus Simulator, which bundles the standalone Venus simulator.

The latter extension is crucial because it allows easily executing and debugging RISC-V assembly in a stepwise manner. Its page on the Visual Studio Marketplace further provides a lot of useful information on its features and how to use them, alongside fun examples.

You may also find the Venus simulator hosted online:

- https://venus.cs61c.org
- https://venus.kvakil.me

We will focus on the VSCode interface, but the online version shares many of the same features.

Resources

You may find the following external resources useful while learning to program in RISC-V assembly:

- The latest official manual for the RISC-V ISA can be found at https://riscv.org/technical/specifications under the PDF link for *Volume 1, Unprivileged Specification*.
- 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 at https://riscv-programming.org/book/riscv-book.html.

Toy example

RISC-V files are usually given the `.s` file extension, which is common to other assembly languages as well.

Consider the following file contents, in this case written to a file named `focaccia.s`:

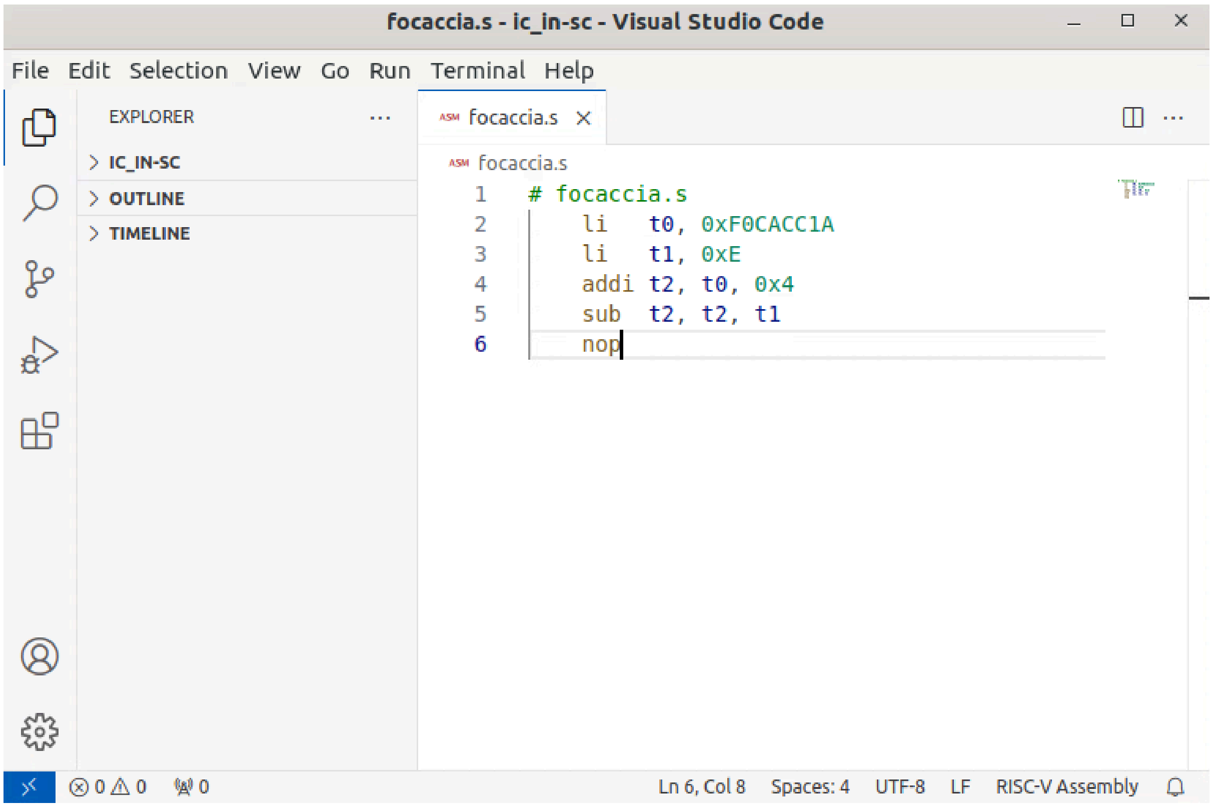
```
# focaccia.s
    li   t0, 0xF0CACC1A
    li   t1, 0xE
    addi t2, t0, 0x4
    sub  t2, t2, t1
    nop
```

This little program first writes a couple of integer literals (*immediate values*) to temporary registers `t0` and `t1`. It then performs a sequence of two arithmetic operations, each time writing the result to a third temporary register, `t2`. Finally, the program ends with an empty *no-operation* instruction, which will come in handy as a placeholder later on.

We expect that the final result in `t2` will be the value `0xF0CACC10`:

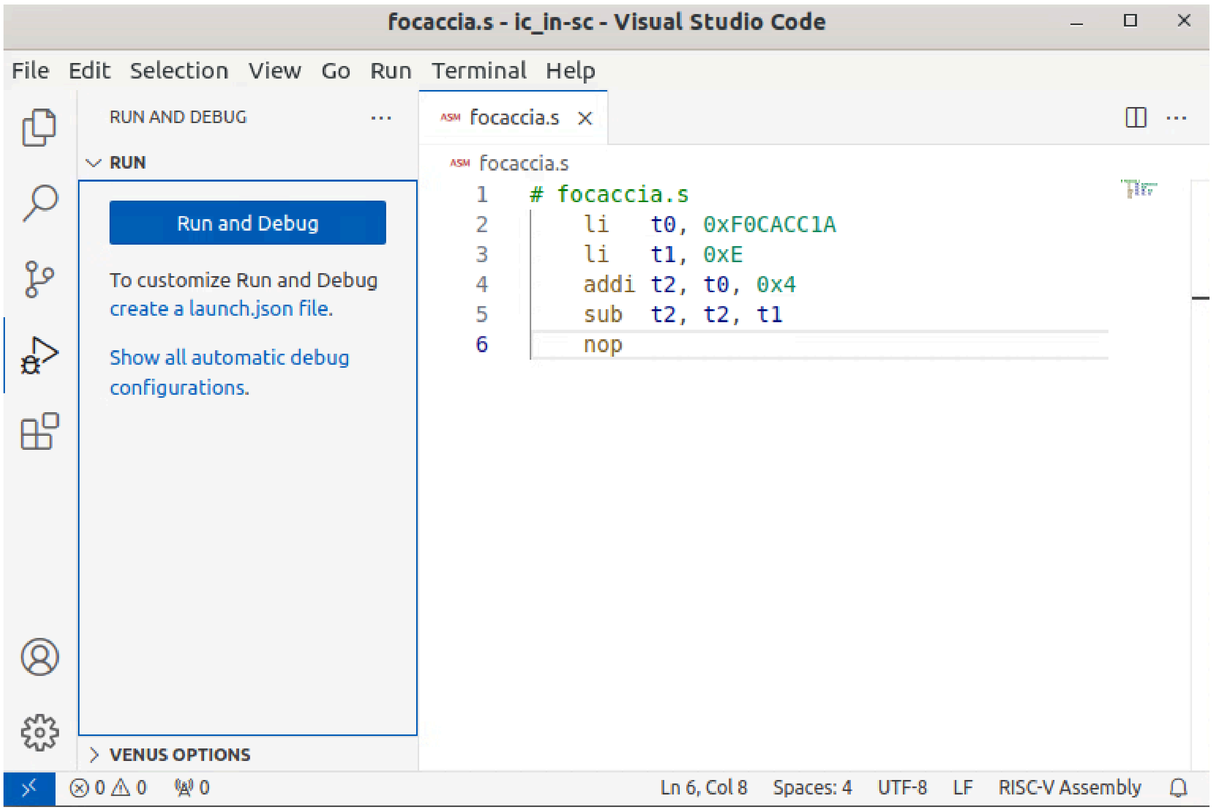
1. The `addition` adds `0x4` to `0xF0CACC1A`, yielding `0xF0CACC1E`.
2. The `subtraction` subtracts `0xE` from the previous result, and `0xE` - `0xE` = `0x0` in the low nibble, yielding `0xF0CACC10`.

Let's make sure the final and intermediate values are indeed as we expect, by entering the program into VSCode:


focaccia.s - ic_in-sc - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
IC_IN-SC
OUTLINE
TIMELINE
focaccia.s
Ln 6, Col 8 Spaces: 4 UTF-8 LF RISC-V Assembly

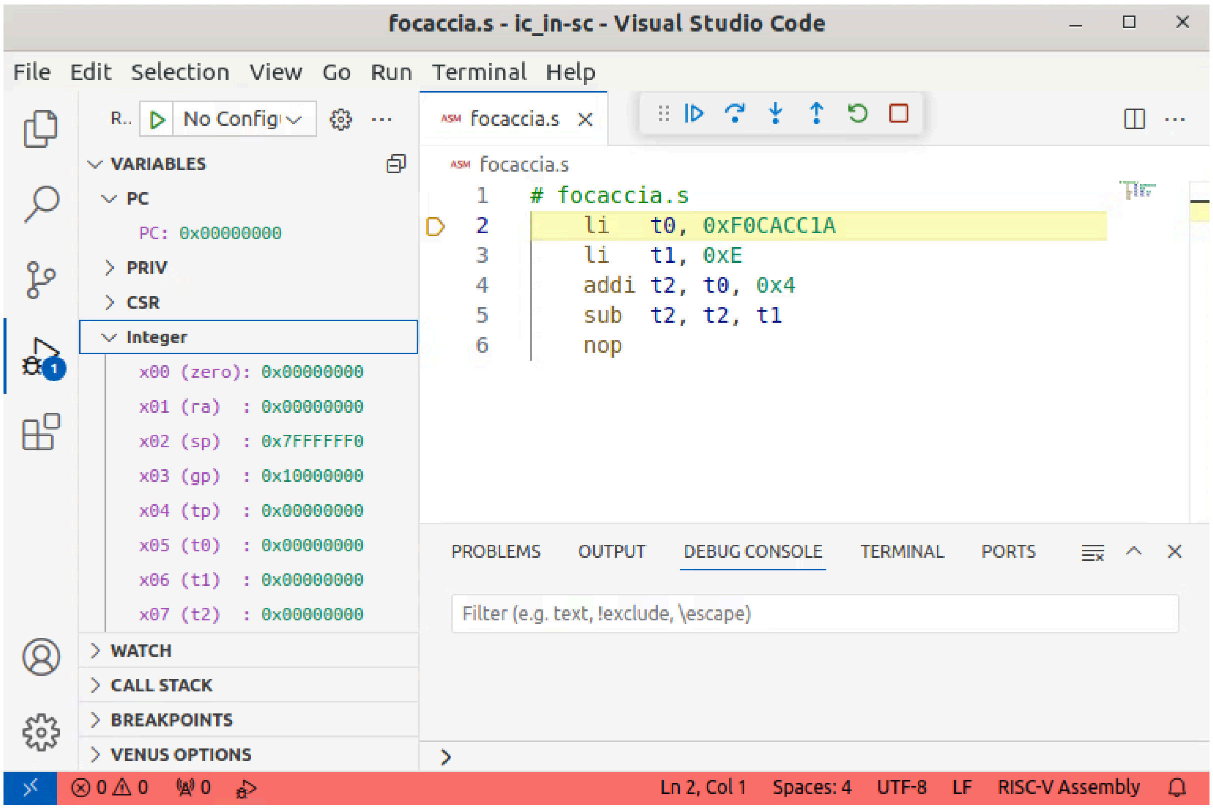
```
# focaccia.s
    li   t0, 0xF0CACC1A
    li   t1, 0xE
    addi t2, t0, 0x4
    sub  t2, t2, t1
    nop
```

Now click on the *Run and Debug* option in the left sidebar, followed by the button of the same name (if present):


focaccia.s - ic_in-sc - Visual Studio Code
File Edit Selection View Go Run Terminal Help
RUN AND DEBUG
RUN
Run and Debug
To customize Run and Debug create a launch.json file.
Show all automatic debug configurations.
VENUS OPTIONS
focaccia.s
Ln 6, Col 8 Spaces: 4 UTF-8 LF RISC-V Assembly

```
# focaccia.s
    li   t0, 0xF0CACC1A
    li   t1, 0xE
    addi t2, t0, 0x4
    sub  t2, t2, t1
    nop
```

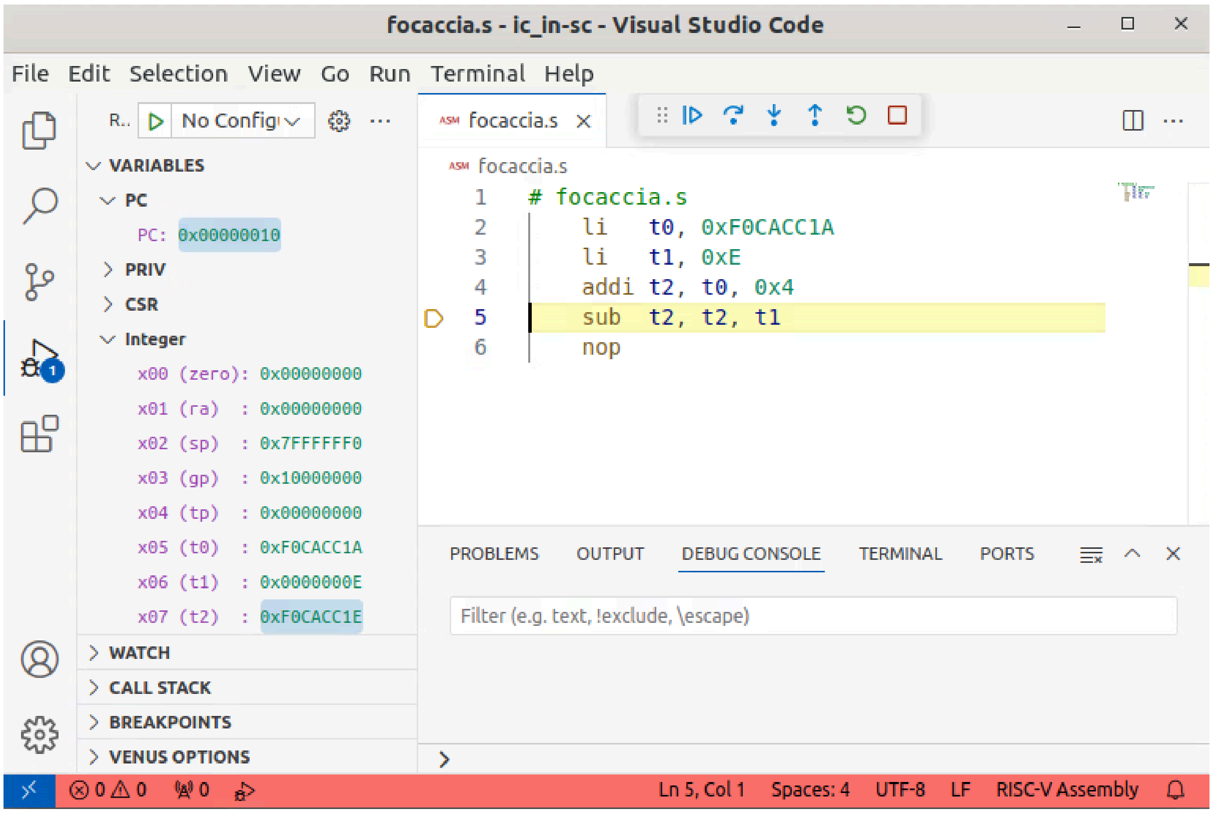
This will start an interactive debugging session, where instructions are executed only on request. Let's expand the *VARIABLES* > *Integer* subsection on the left, so that we can track changes to register values as the program runs:


focaccia.s - ic_in-sc - Visual Studio Code
File Edit Selection View Go Run Terminal Help
No Config
focaccia.s
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Filter (e.g. text, !exclude, \escape)
WATCH
CALL STACK
BREAKPOINTS
VENUS OPTIONS
Ln 2, Col 1 Spaces: 4 UTF-8 LF RISC-V Assembly

```
VARIABLES
  PC
    PC: 0x00000000
  PRIV
  CSR
  Integer
    x00 (zero): 0x00000000
    x01 (ra)  : 0x00000000
    x02 (sp)  : 0x7FFFFFF0
    x03 (gp)  : 0x10000000
    x04 (tp)  : 0x00000000
    x05 (t0)  : 0x00000000
    x06 (t1)  : 0x00000000
    x07 (t2)  : 0x00000000
```

```
# focaccia.s
    li   t0, 0xF0CACC1A
    li   t1, 0xE
    addi t2, t0, 0x4
    sub  t2, t2, t1
    nop
```

Now we can click the *Step Over* button (with the curved arrow) in the small ephemeral panel at the top three times, until we reach the point right after the `addi` instruction has been executed:

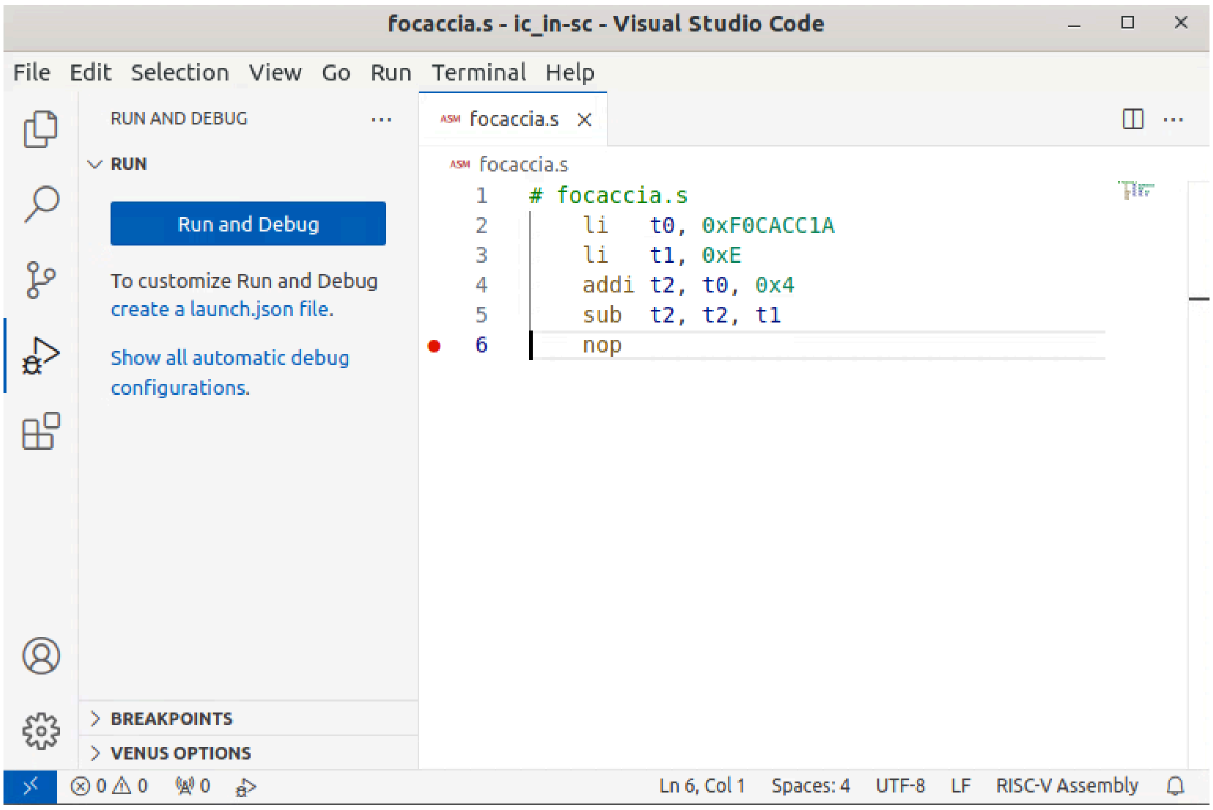

focaccia.s - ic_in-sc - Visual Studio Code
File Edit Selection View Go Run Terminal Help
No Config
focaccia.s
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Filter (e.g. text, !exclude, \escape)
WATCH
CALL STACK
BREAKPOINTS
VENUS OPTIONS
Ln 5, Col 1 Spaces: 4 UTF-8 LF RISC-V Assembly

```
VARIABLES
  PC
    PC: 0x00000010
  PRIV
  CSR
  Integer
    x00 (zero): 0x00000000
    x01 (ra)  : 0x00000000
    x02 (sp)  : 0x7FFFFFF0
    x03 (gp)  : 0x10000000
    x04 (tp)  : 0x00000000
    x05 (t0)  : 0xF0CACC1A
    x06 (t1)  : 0x0000000E
    x07 (t2)  : 0xF0CACC1E
```

```
# focaccia.s
    li   t0, 0xF0CACC1A
    li   t1, 0xE
    addi t2, t0, 0x4
    sub  t2, t2, t1
    nop
```

Indeed we see that the intermediate value in register `t2` is the expected `0xF0CACC1E`, and we can terminate execution by clicking one of the buttons *Continue* or *Stop* in the same panel as the *Step Over* button.

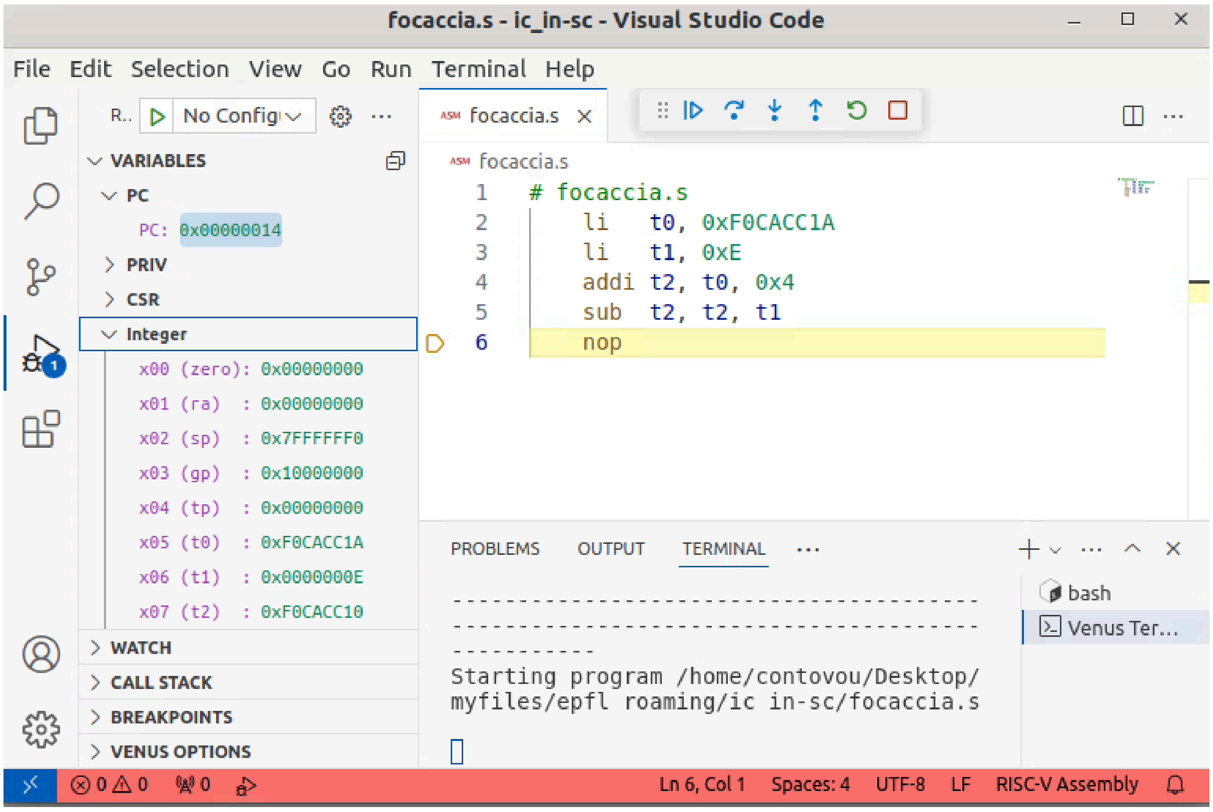
It may not always be practical to step over individual instructions, however. In this example it would be nice to skip past the execution of sub, and inspect only the final value in t2. To do this, we can set a breakpoint on the final nop instruction by hovering the mouse to the left of the line number, and clicking on the red dot that appears:



focaccia.s - ic_in-sc - Visual Studio Code
File Edit Selection View Go Run Terminal Help
RUN AND DEBUG
RUN
Run and Debug
To customize Run and Debug create a launch.json file.
Show all automatic debug configurations.
BREAKPOINTS
VENUS OPTIONS
focaccia.s
Ln 6, Col 1 Spaces: 4 UTF-8 LF RISC-V Assembly

```
# focaccia.s
    li   t0, 0xF0CACC1A
    li   t1, 0xE
    addi t2, t0, 0x4
    sub  t2, t2, t1
    nop
```

Now we can restart the *Run and Debug* session, and subsequently click the *Continue* button a single time. This should execute all instructions up to but not including the nop instruction on which we set the breakpoint. At this point, the preceding sub instruction has just been executed, and we can confirm that the final value of t2 is the expected 0xF0CACC10:


focaccia.s - ic_in-sc - Visual Studio Code
File Edit Selection View Go Run Terminal Help
No Config
focaccia.s
VARIABLES
PC
PRIV
CSR
Integer
WATCH
CALL STACK
BREAKPOINTS
VENUS OPTIONS
PROBLEMS OUTPUT TERMINAL
bash
Venus Ter...
Ln 6, Col 1 Spaces: 4 UTF-8 LF RISC-V Assembly

```
PC: 0x00000014
x00 (zero): 0x00000000
x01 (ra)  : 0x00000000
x02 (sp)  : 0x7FFFFFF0
x03 (gp)  : 0x10000000
x04 (tp)  : 0x00000000
x05 (t0)  : 0xF0CACC1A
x06 (t1)  : 0x0000000E
x07 (t2)  : 0xF0CACC10
```

```
# focaccia.s
    li   t0, 0xF0CACC1A
    li   t1, 0xE
    addi t2, t0, 0x4
    sub  t2, t2, t1
    nop
```

```
----------------------------------------
----------------------------------------
------------
Starting program /home/contovou/Desktop/
myfiles/epfl roaming/ic in-sc/focaccia.s
```

Memory view

RISC-V computations always involve registers, so it is useful to know how to manipulate them and keep track of their values as a program runs. The limited number of registers cannot, however, entirely fit a realistic program's data, so we also need to be able to manipulate and inspect the system's memory. Luckily, the Venus simulator provides a convenient *memory view* which displays the contents of memory at different addresses in real time.

Memory is modelled as one large contiguous array of bytes, which is a uniform (and thus convenient) representation when writing programs. But we would like to store in it data of different natures: for example, to keep program instructions separate from the data that they consume. Thus it is convenient, especially for the system, to partition the memory into different *segments* or *sections*.

In the following example, we will use two common sections: one for holding program instructions (identified by the assembler directive `.text`), and another for holding *static* data that we know ahead of time, i.e. before the program is even run (identified by the assembler directive `.data`). Venus places these sections at different locations in memory: program instructions in `.text` start at address `0x00000000`, whereas `.data` starts at `0x10000000`.

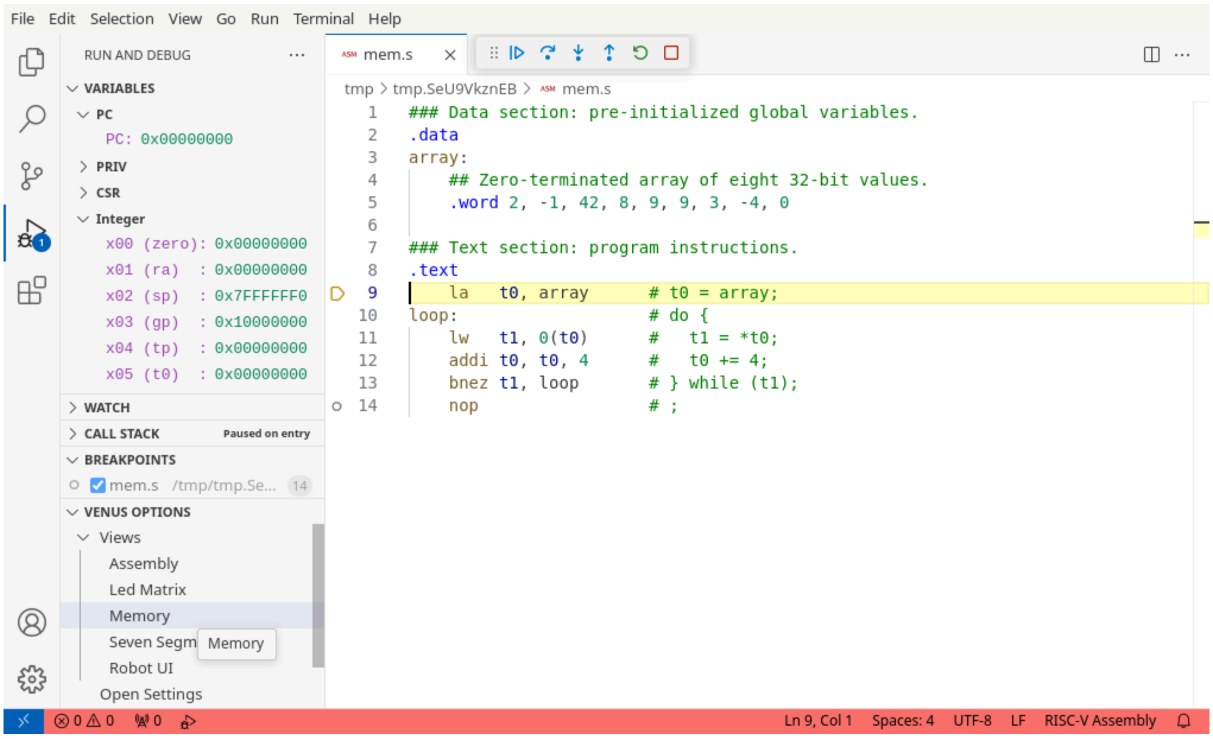
Let's now see how it looks in practice. Consider the following simple program:

```
### Data section: pre-initialized global variables.
.data
array:
    ## Zero-terminated array of eight 32-bit values.
    .word 2, -1, 42, 8, 9, 9, 3, -4, 0

### Text section: program instructions.
.text
    la   t0, array      # t0 = array;
loop:                   # do {
    lw   t1, 0(t0)      #   t1 = *t0;
    addi t0, t0, 4      #   t0 += 4;
    bnez t1, loop       # } while (t1);
    nop                 # ;
```

First, it reserves a contiguous block of nine 32-bit *words* in the `.data` section, and initialises the block with the given numbers. Then, when the program in the `.text` section is executed, it finds the start of the static array, and keeps advancing over its elements until it encounters a zero.

Let's enter this program into VSCode, and add a breakpoint as before on the final line with the nop instruction. Again, click on *Run and Debug* and expand the *VARIABLES > Integer* subsection. But before continuing execution, let's open the memory view. At the bottom of the left sidebar is a subsection calling *VENUS OPTIONS*. Find the *VENUS OPTIONS > Views > Memory* suboption:

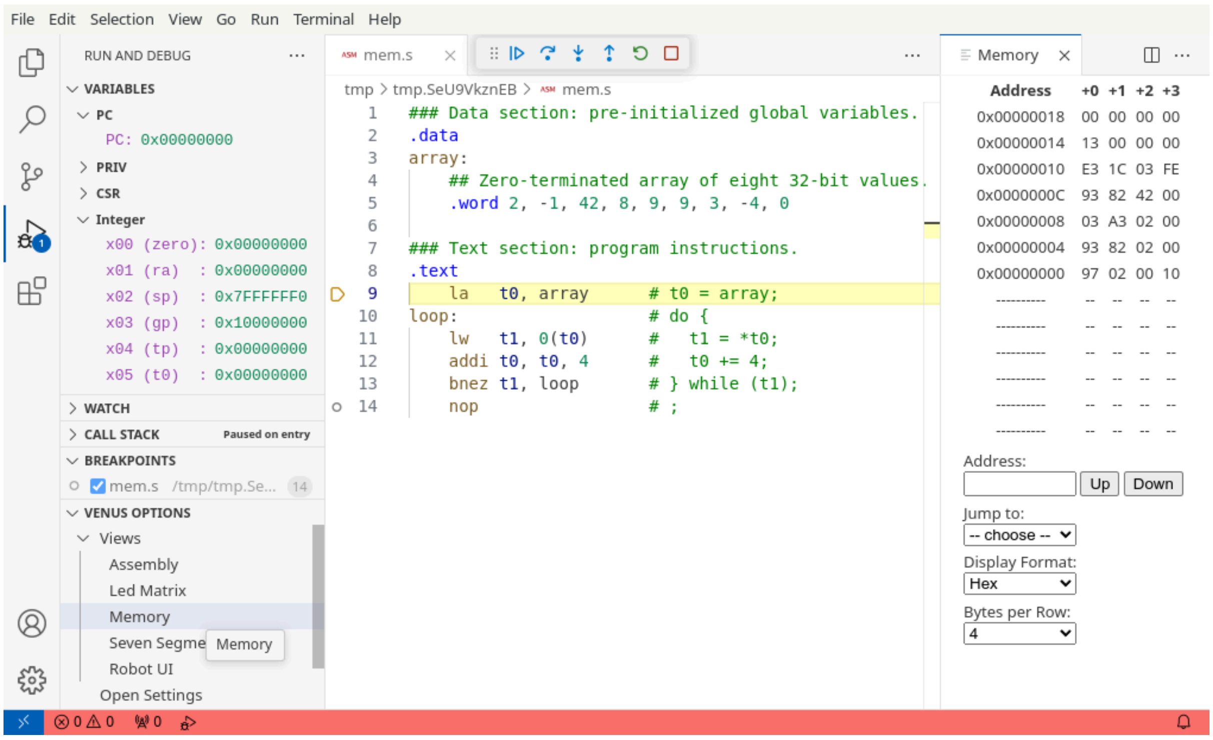

File Edit Selection View Go Run Terminal Help
RUN AND DEBUG
mem.s
tmp > tmp.SeU9VkznEB > mem.s
WATCH
CALL STACK Paused on entry
BREAKPOINTS
mem.s /tmp/tmp.Se... 14
VENUS OPTIONS
Views
Assembly
Led Matrix
Memory
Seven Segm
Memory
Robot UI
Open Settings
Ln 9, Col 1 Spaces: 4 UTF-8 LF RISC-V Assembly

```
VARIABLES
  PC
    PC: 0x00000000
  PRIV
  CSR
  Integer
    x00 (zero): 0x00000000
    x01 (ra)  : 0x00000000
    x02 (sp)  : 0x7FFFFFF0
    x03 (gp)  : 0x10000000
    x04 (tp)  : 0x00000000
    x05 (t0)  : 0x00000000
```

```
### Data section: pre-initialized global variables.
.data
array:
    ## Zero-terminated array of eight 32-bit values.
    .word 2, -1, 42, 8, 9, 9, 3, -4, 0

### Text section: program instructions.
.text
    la   t0, array      # t0 = array;
loop:                   # do {
    lw   t1, 0(t0)      #   t1 = *t0;
    addi t0, t0, 4      #   t0 += 4;
    bnez t1, loop       # } while (t1);
    nop                 # ;
```

This should pop up a new tab showing the contents of memory and their addresses:



```
VARIABLES
  PC
    PC: 0x00000000
  PRIV
  CSR
  Integer
    x00 (zero): 0x00000000
    x01 (ra)  : 0x00000000
    x02 (sp)  : 0x7FFFFFF0
    x03 (gp)  : 0x10000000
    x04 (tp)  : 0x00000000
    x05 (t0)  : 0x00000000
```

```
### Data section: pre-initialized global variables.
.data
array:
    ## Zero-terminated array of eight 32-bit values.
    .word 2, -1, 42, 8, 9, 9, 3, -4, 0

### Text section: program instructions.
.text
    la   t0, array       # t0 = array;
loop:                    # do {
    lw   t1, 0(t0)       #   t1 = *t0;
    addi t0, t0, 4       #   t0 += 4;
    bnez t1, loop        # } while (t1);
    nop                  # ;
```

Address	+0	+1	+2	+3
0x00000018	00	00	00	00
0x00000014	13	00	00	00
0x00000010	E3	1C	03	FE
0x0000000C	93	82	42	00
0x00000008	03	A3	02	00
0x00000004	93	82	02	00
0x00000000	97	02	00	10
----------	--	--	--	--
----------	--	--	--	--
----------	--	--	--	--
----------	--	--	--	--
----------	--	--	--	--
----------	--	--	--	--

Address:
Jump to: -- choose --
Display Format: Hex
Bytes per Row: 4

By default the view starts at address `0x00000000`, where our program instructions are stored. You should see six word-size instructions at addresses `0x00000000` through `0x00000014`, after which the contents of memory become zero. In ascending address order, and reading bytes from right to left to account for little-endianness, we see the following instructions:

1. `0x10000297`
2. `0x00028293`
3. `0x0002A303`
4. `0x00428293`
5. `0xFE031CE3`
6. `0x00000013`

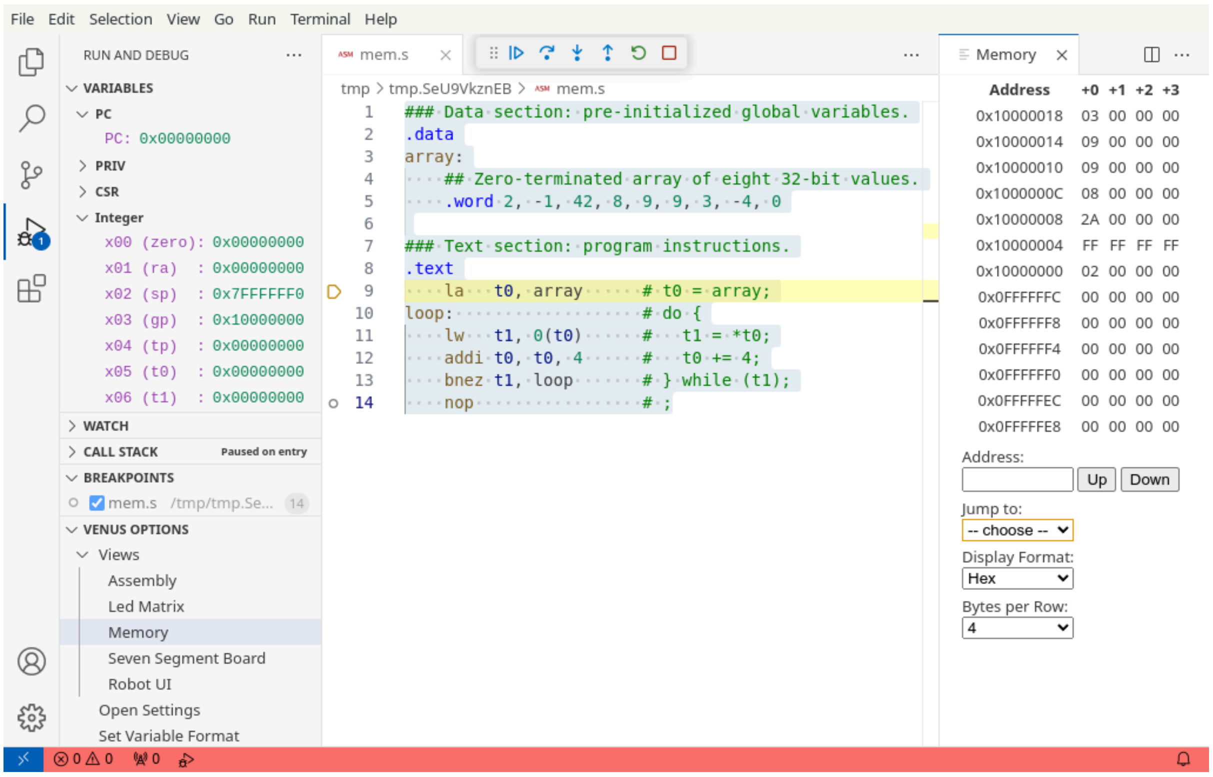
The first instruction in our program, `la t0, array`, is actually a higher-level pseudoinstruction which the assembler converts into two lower-level instructions: `auipc x5 0x10000` and `addi x5 x5 0` (temporary register `t0` is an alias for `x5`). `auipc` adds a 20-bit immediate value to the most significant bits of the program counter (PC), storing the result in a register. This is followed by an addition of the remaining lower 12 bits, which in this case are all zero. Taken together, the result is that `t0` now contains `0x10000` concatenated with zeros to form `0x10000000`, the start address of the `.data` section, and by extension the start address of our `array`.

The instruction `0x10000297` is encoded as follows: the 20 most significant bits correspond to the immediate value `0x10000`; these are followed by the five bits `00101` identifying the destination register as `x5`; and the last seven bits `0010111` correspond to the instruction `auipc`.

Similarly, `0x00028293` breaks down as an immediate zero in the 12 most significant bits; followed by the five bits `00101` identifying the source register `x5`; then three zero bits; then the destination register `00101` again; and finally the `addi` opcode, `0010011`.

The rest of the instructions are encoded similarly, and you are encouraged to work through them together with the RISC-V ISA manual.

Now let's look at the contents of `array` in the `.data` section. In the memory view on the right, click on the *Jump to* dropdown menu and select *Data*:



```
VARIABLES
  PC
    PC: 0x00000000
  PRIV
  CSR
  Integer
    x00 (zero): 0x00000000
    x01 (ra)  : 0x00000000
    x02 (sp)  : 0x7FFFFFF0
    x03 (gp)  : 0x10000000
    x04 (tp)  : 0x00000000
    x05 (t0)  : 0x00000000
    x06 (t1)  : 0x00000000
```

```
### Data section: pre-initialized global variables.
.data
array:
    ## Zero-terminated array of eight 32-bit values.
    .word 2, -1, 42, 8, 9, 9, 3, -4, 0

### Text section: program instructions.
.text
    la   t0, array      # t0 = array;
loop:                   # do {
    lw   t1, 0(t0)      #   t1 = *t0;
    addi t0, t0, 4      #   t0 += 4;
    bnez t1, loop       # } while (t1);
    nop                 # ;
```

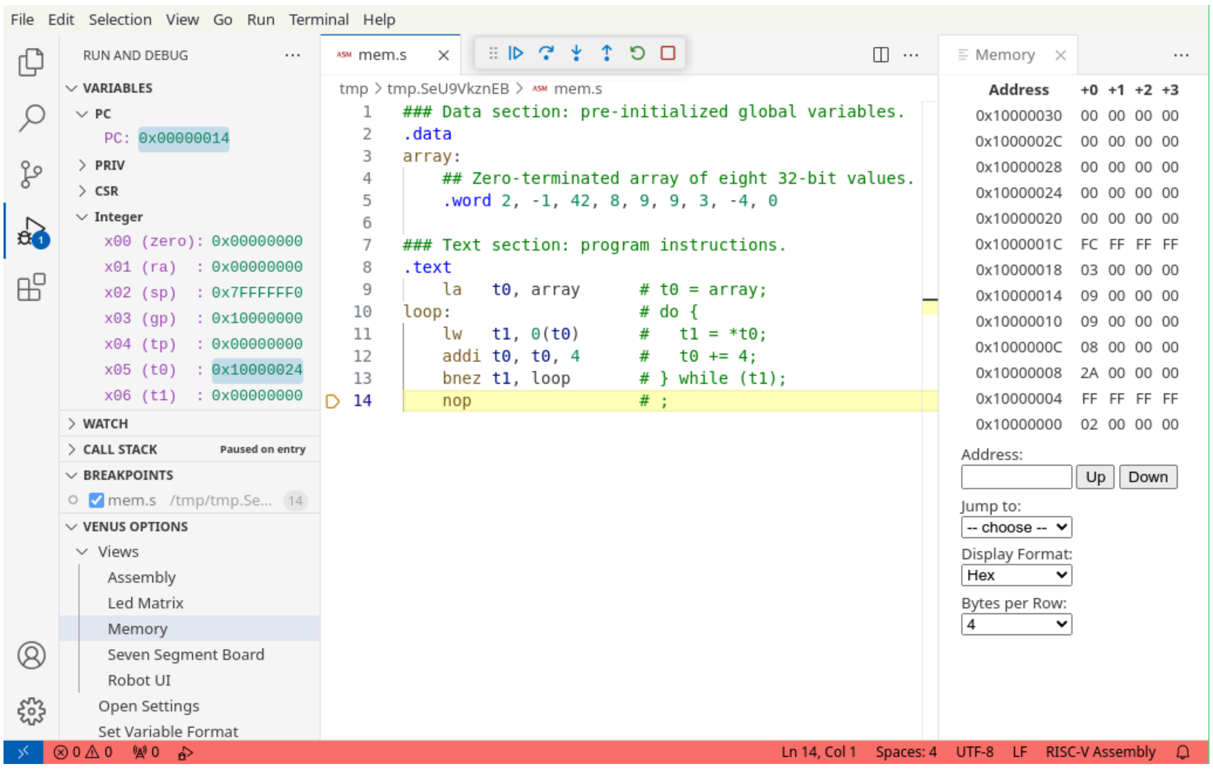
Address	+0	+1	+2	+3
0x10000018	03	00	00	00
0x10000014	09	00	00	00
0x10000010	09	00	00	00
0x1000000C	08	00	00	00
0x10000008	2A	00	00	00
0x10000004	FF	FF	FF	FF
0x10000000	02	00	00	00
0x0FFFFFFC	00	00	00	00
0x0FFFFFF8	00	00	00	00
0x0FFFFFF4	00	00	00	00
0x0FFFFFF0	00	00	00	00
0x0FFFFFEC	00	00	00	00
0x0FFFFFE8	00	00	00	00

Once again, in ascending order, and reading bytes from right to left, we see the following values starting at address `0x10000000`:

1. `0x00000002`
2. `0xFFFFFFFF`
3. `0x0000002A`
4. `0x00000008`
5. …

and so on, ending with `0x00000000` at address `0x10000020`. This is, of course, our array of 32-bit numbers: 2, -1 (in two's complement), 42, etc.

Now let's run the program by clicking the *Continue* button in the debugging panel at the top:



```
PC
  PC: 0x00000014
PRIV
CSR
Integer
  x00 (zero): 0x00000000
  x01 (ra)  : 0x00000000
  x02 (sp)  : 0x7FFFFFF0
  x03 (gp)  : 0x10000000
  x04 (tp)  : 0x00000000
  x05 (t0)  : 0x10000024
  x06 (t1)  : 0x00000000
```

```
### Data section: pre-initialized global variables.
.data
array:
    ## Zero-terminated array of eight 32-bit values.
    .word 2, -1, 42, 8, 9, 9, 3, -4, 0

### Text section: program instructions.
.text
    la    t0, array      # t0 = array;
loop:                    # do {
    lw    t1, 0(t0)      #   t1 = *t0;
    addi  t0, t0, 4      #   t0 += 4;
    bnez  t1, loop       # } while (t1);
    nop                  # ;
```

Address	+0	+1	+2	+3
0x10000030	00	00	00	00
0x1000002C	00	00	00	00
0x10000028	00	00	00	00
0x10000024	00	00	00	00
0x10000020	00	00	00	00
0x1000001C	FC	FF	FF	FF
0x10000018	03	00	00	00
0x10000014	09	00	00	00
0x10000010	09	00	00	00
0x1000000C	08	00	00	00
0x10000008	2A	00	00	00
0x10000004	FF	FF	FF	FF
0x10000000	02	00	00	00

Display Format: Hex
Bytes per Row: 4

At this point, the loop has iterated over all elements in `array`, and stopped at the trailing zero. We can see the results in the two temporary registers used: `t0` contains the address `0x10000024` which, as we just saw, is 4 bytes past the end of `array`; and `t1` contains `0x00000000`, the last value in `array`.