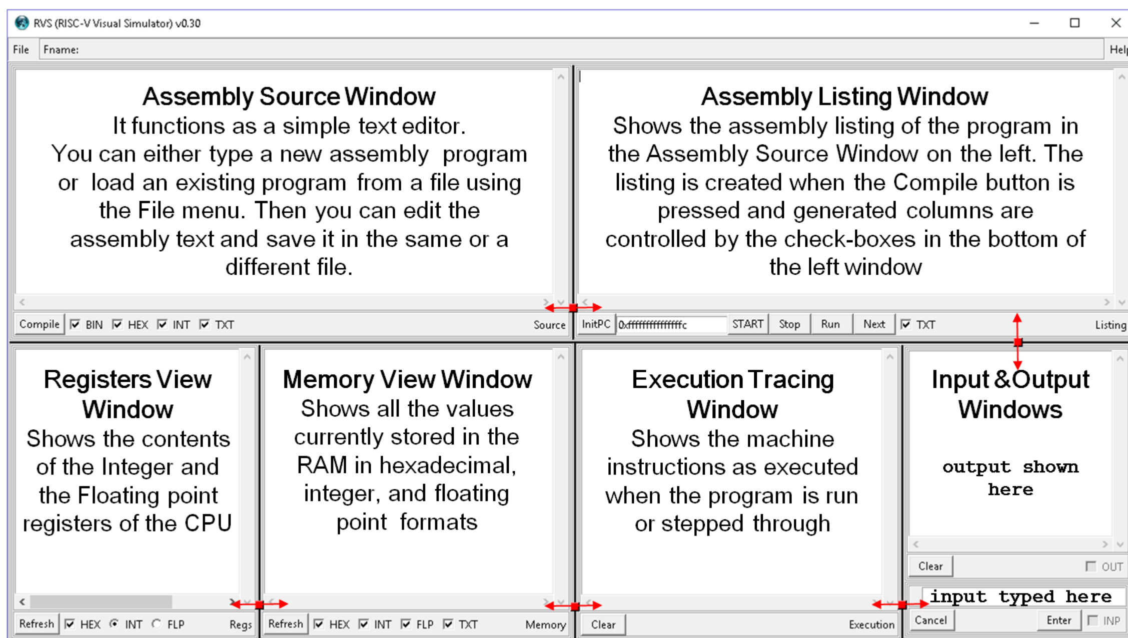
RVS (RISC-V Visual Simulator)

User Manual v0.07

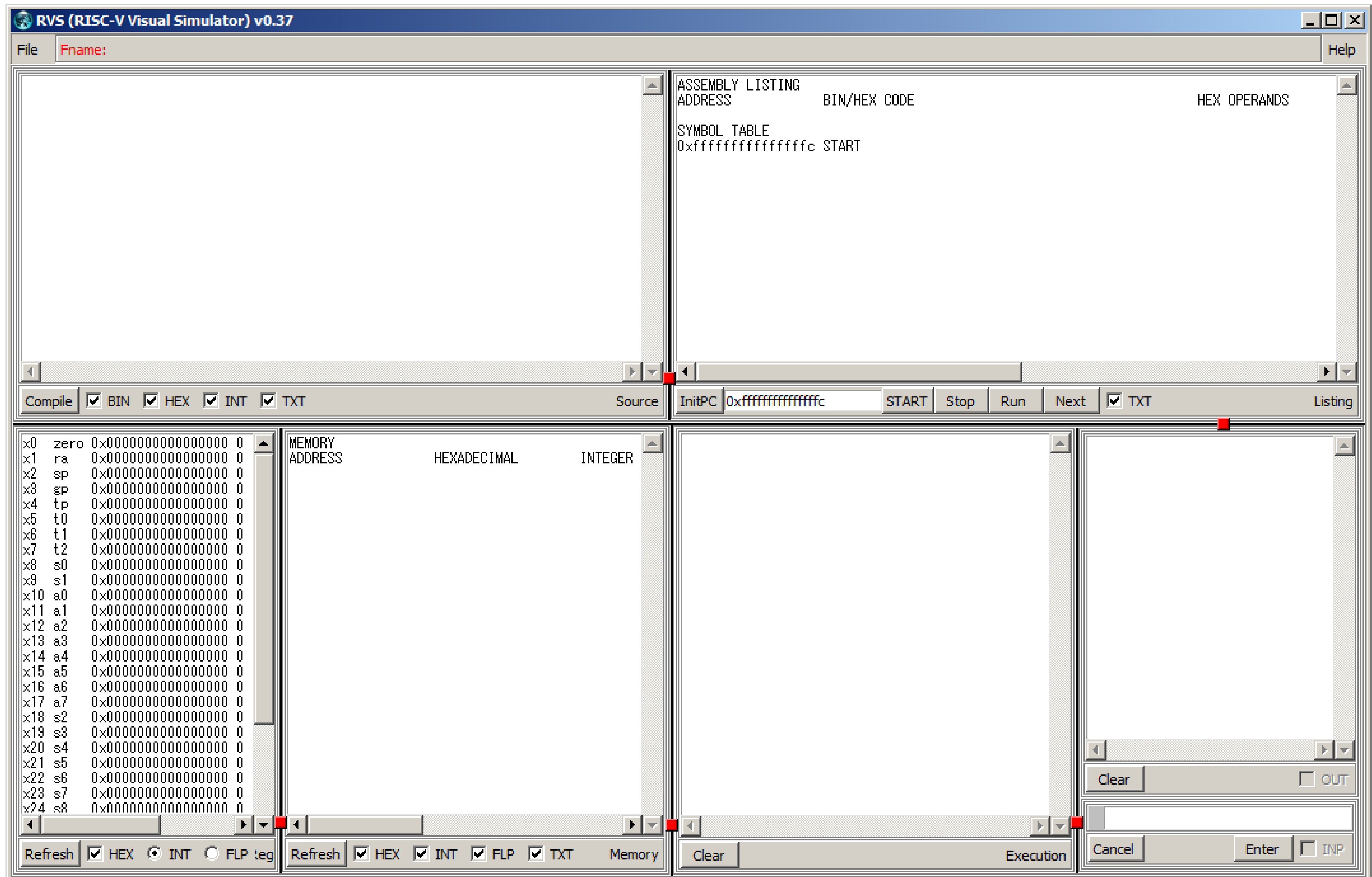
1. Outline

The Risk-V Visual Simulator (RVS) provides a convenient Graphical User Interface (GUI) for input, editing, assembly, and execution tracing of RISC-V assembly programs. RVS supports the subset of RISC-V instructions in the book "Computer Organization and Design: The Hardware/Software Interface, RISC-V Edition" by Patterson and Hennessey. The GUI consists of six windows as shown below.

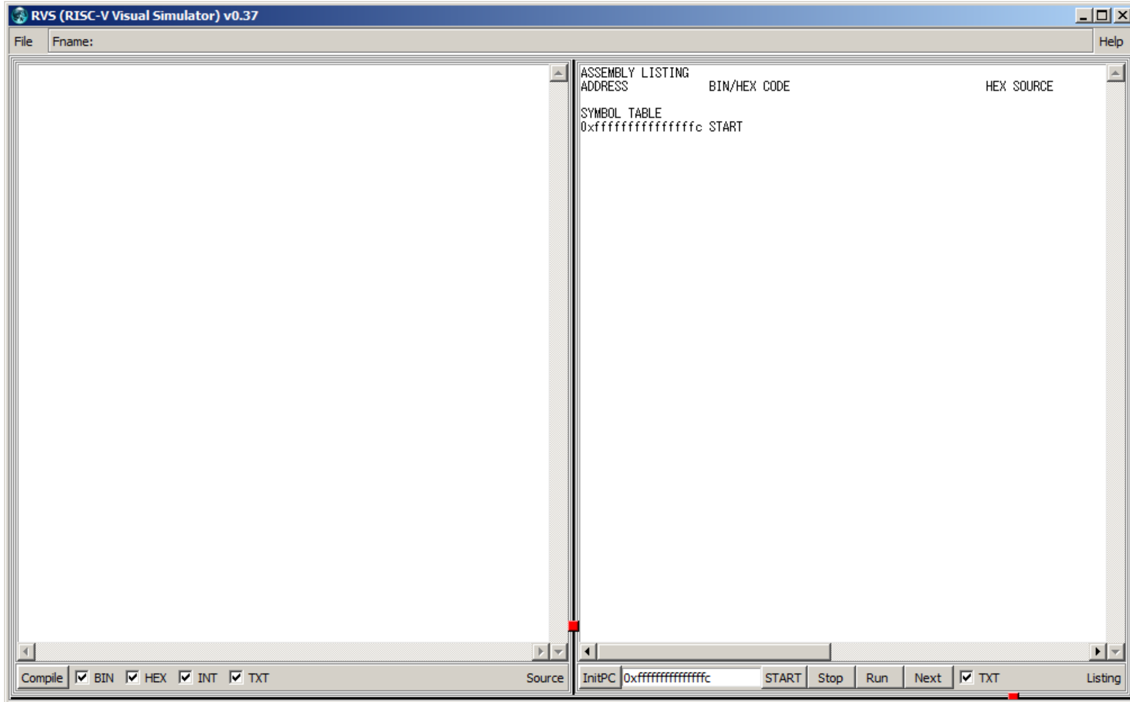


The minimal width and height of the main window is restricted to ensure that all controls (buttons, check boxes, labels, etc.) are visible. The area of each window is freely adjustable by moving the five handles shown in red in the above snapshot as indicated by the arrows superimposed on them. Note that there are no explicit controls to selectively hide/show individual windows because this can be easily done with the red handles.

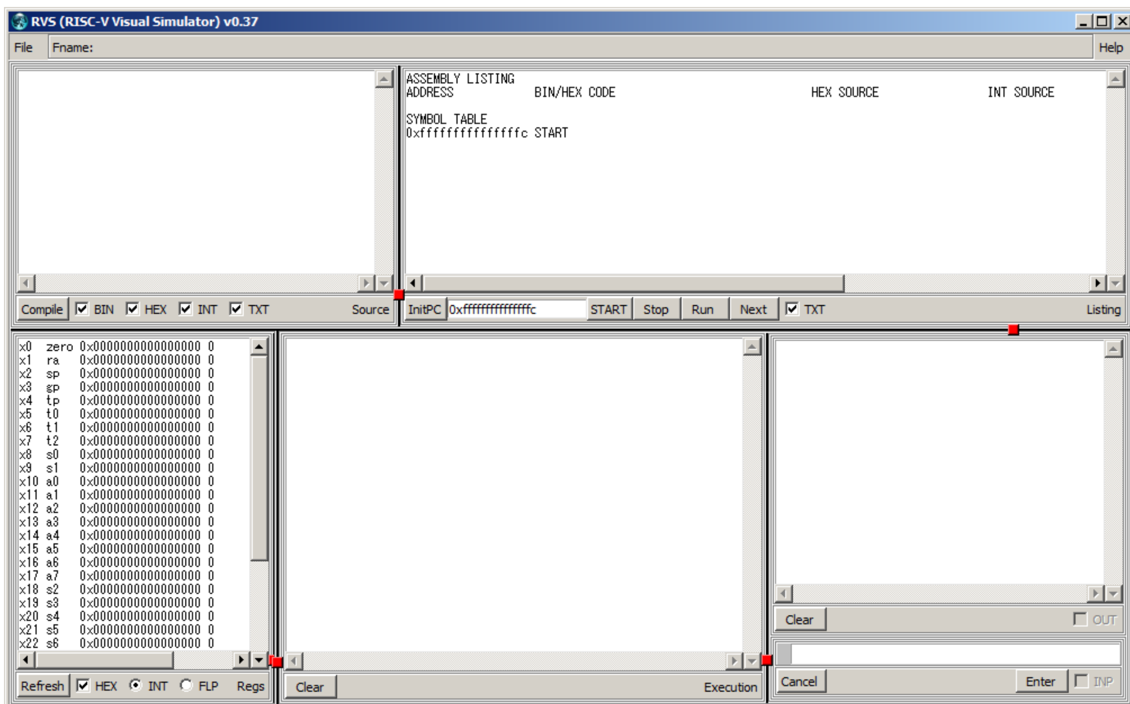
In the following snapshots we show the initial window as it looks at startup.


RVS (RISC-V Visual Simulator) v0.37
File
Fname:
Help
ASSEMBLY LISTING
ADDRESS BIN/HEX CODE HEX OPERANDS
SYMBOL TABLE
0xfffffffffffffffc START
Compile BIN HEX INT TXT Source
InitPC 0xfffffffffffffffc START Stop Run Next TXT Listing
x0 zero 0x0000000000000000 0
x1 ra 0x0000000000000000 0
x2 sp 0x0000000000000000 0
x3 gp 0x0000000000000000 0
x4 tp 0x0000000000000000 0
x5 t0 0x0000000000000000 0
x6 t1 0x0000000000000000 0
x7 t2 0x0000000000000000 0
x8 s0 0x0000000000000000 0
x9 s1 0x0000000000000000 0
x10 a0 0x0000000000000000 0
x11 a1 0x0000000000000000 0
x12 a2 0x0000000000000000 0
x13 a3 0x0000000000000000 0
x14 a4 0x0000000000000000 0
x15 a5 0x0000000000000000 0
x16 a6 0x0000000000000000 0
x17 a7 0x0000000000000000 0
x18 s2 0x0000000000000000 0
x19 s3 0x0000000000000000 0
x20 s4 0x0000000000000000 0
x21 s5 0x0000000000000000 0
x22 s6 0x0000000000000000 0
x23 s7 0x0000000000000000 0
MEMORY
ADDRESS HEXADECIMAL INTEGER
Clear OUT
Refresh HEX INT FLP
Refresh HEX INT FLP TXT Memory
Clear Execution
Cancel Enter INP

Some window arrangements done through the red handles that might be useful at different staged of the code development are shown below.



Above, only the **Source** and the **Listing** windows are visible. Might be suitable for repeated source editing and recompiling until all syntax errors are corrected.



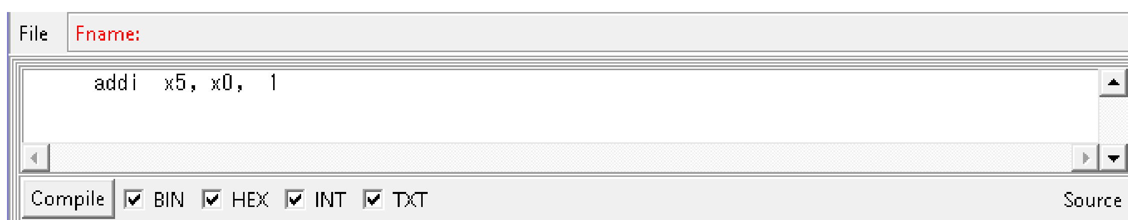
If only data in registers is used, the **Memory** window could be hidden as above.

2. Source code

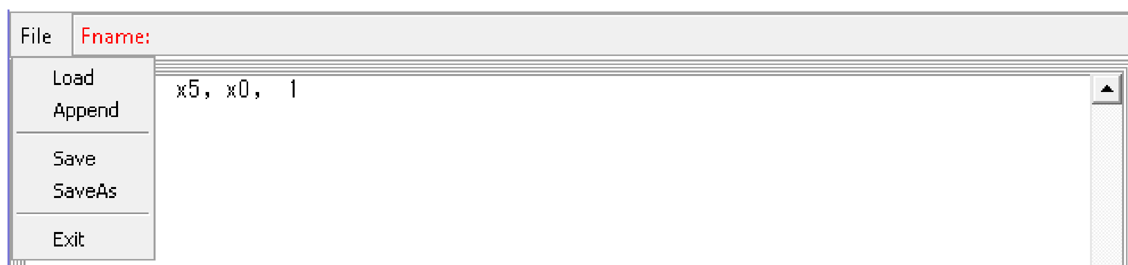
To start, type in the **Source** window the following simple RISC-V assembly instruction:

```
addi  x5, x0,  1
```

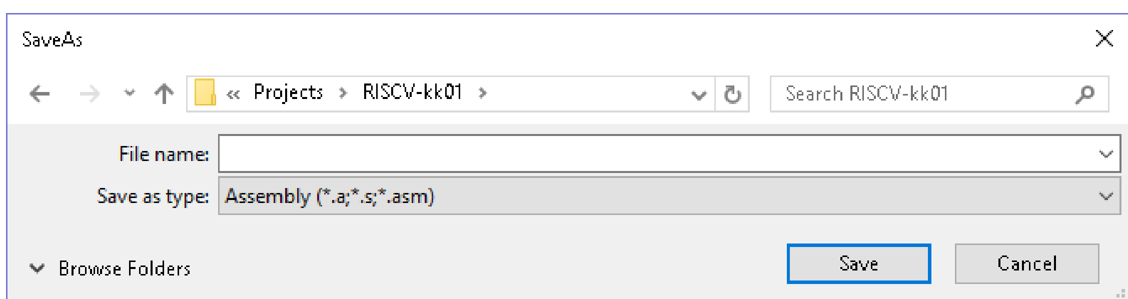
The resulting **Source** window is shown in the following snapshot.



Note that the **Fname:** string next to the **File** menu has turned red. This change in color indicates that you have unsaved changes of the text in the **Source** window. You can save the text from the **Source** window into a file by clicking on the **File** menu as shown below.

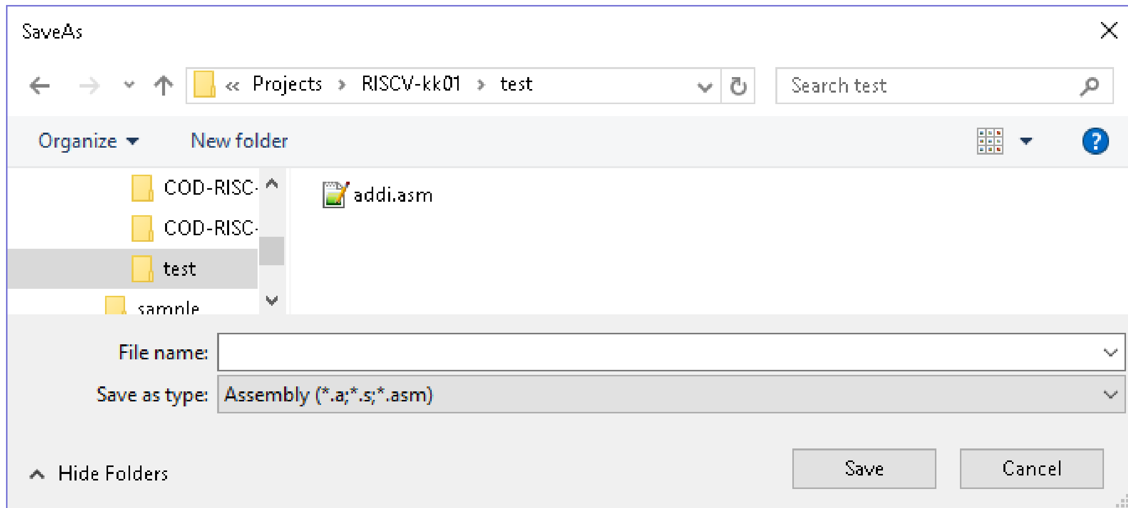


Then select **SaveAs** to obtain the following dialog.

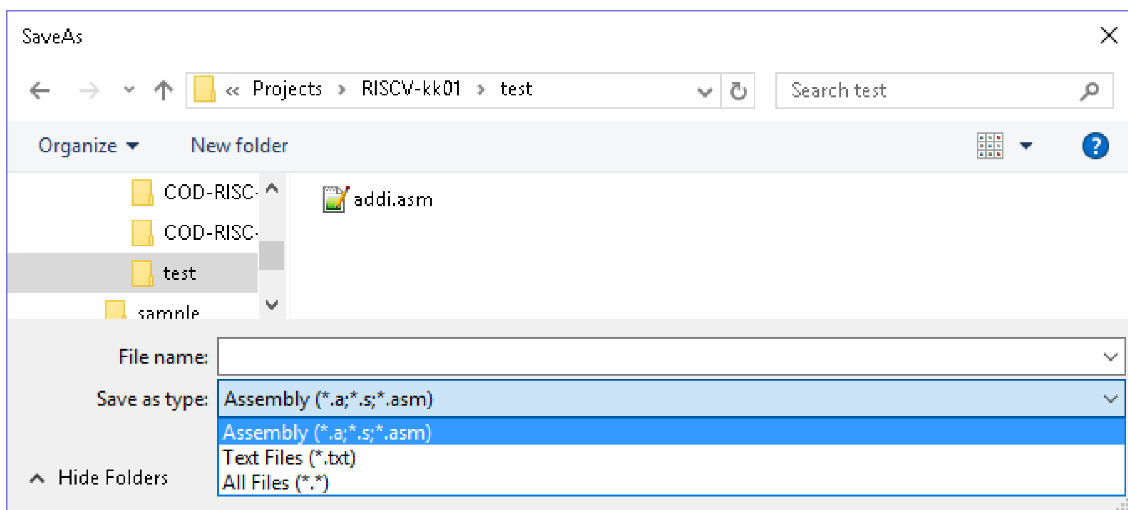


To save the **Source** text in a file, type the file name with its extension (e.g. test.asm) and then click on the **Save** button.

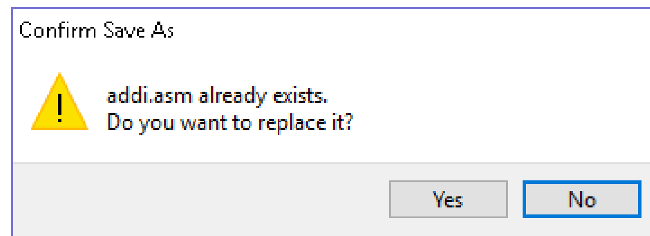
If you want to select a different folder before saving the file, you can click on **Browse Folders** and the above dialog will change as shown in the following snapshot.



The above dialog allows you to change the current directory and see the files in it. Note that only files with extensions shown in the **Save as type:** field, namely .a, .s, and .asm will be visible. To see other files change the **Save as Type:** selection as shown below.

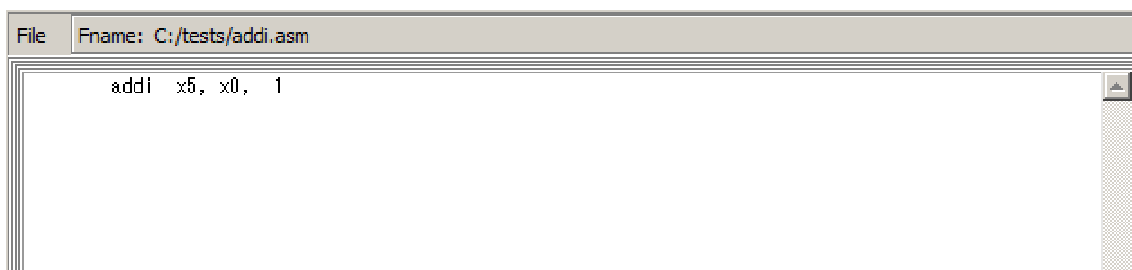


You can see above that a file named **addi.asm** is already present in the current directory. If you try to save your **Source** text under the same name the following warning message will appear.



Clicking on the **Yes** button will overwrite the content of the existing file with your **Source** text.

Once you save your **Source** text in a file, the name of the current file will show after the **Fname:** string.



You can save the **Source** text in the current file at any time by selecting **Save** from the **File** menu. (Note that the **Fname:** string is currently shown in black indicating no unsaved changes.)

The **File** menu also includes two options for loading in the **Source** window existing assembly programs from files. The **Load** option will overwrite the current **Source** text with the new one from the file (a warning will be issued in case of unsaved changes.) The **Append** option on the other hand will add the new program text from the file after the last character of the current **Source** text. Note that after using **Append** the file name will be cleared and the unsaved changes flag will be raised.

The last option of the File menu is Exit which just terminates the RVS program (a warning will be issued in case of unsaved changes.)

For editing the source text, the standard Windows keyboard shortcuts can be used: CTRL+a (select all), CTRL+x (cut), CTRL+c (copy), CTRL+v (paste), CTRL+z (undo), CTRL+y (redo).

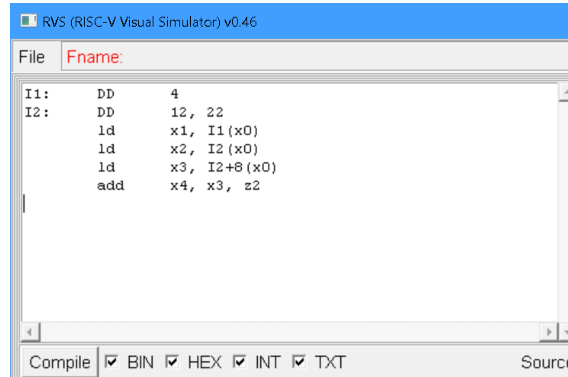
3. Compilation

In this section we will show some compilation examples. Let us begin with entering the following code into the **Source** window:

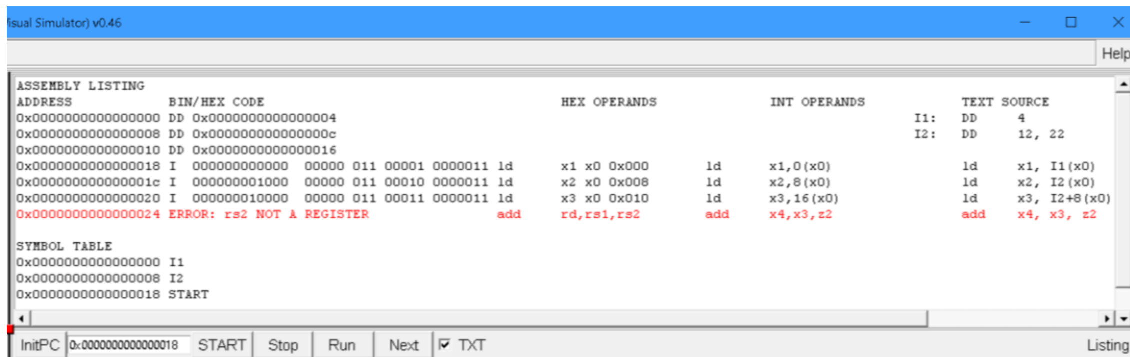
```
I1:     DD      4
I2:     DD      12, 22
        ld      x1, I1(x0)
        ld      x2, I2(x0)
        ld      x3, I2+8(x0)
        add     x4, x3, z2
```

The first 2 lines in the above program employ Define Double (`DD`) assembler commands to store in the memory the three integer constants `4`, `12`, and `22`. The `3` lines that follow employ Load Double (`ld`) RISC-V instructions to load the above values in the registers `x1`, `x2` and `x3` respectively.

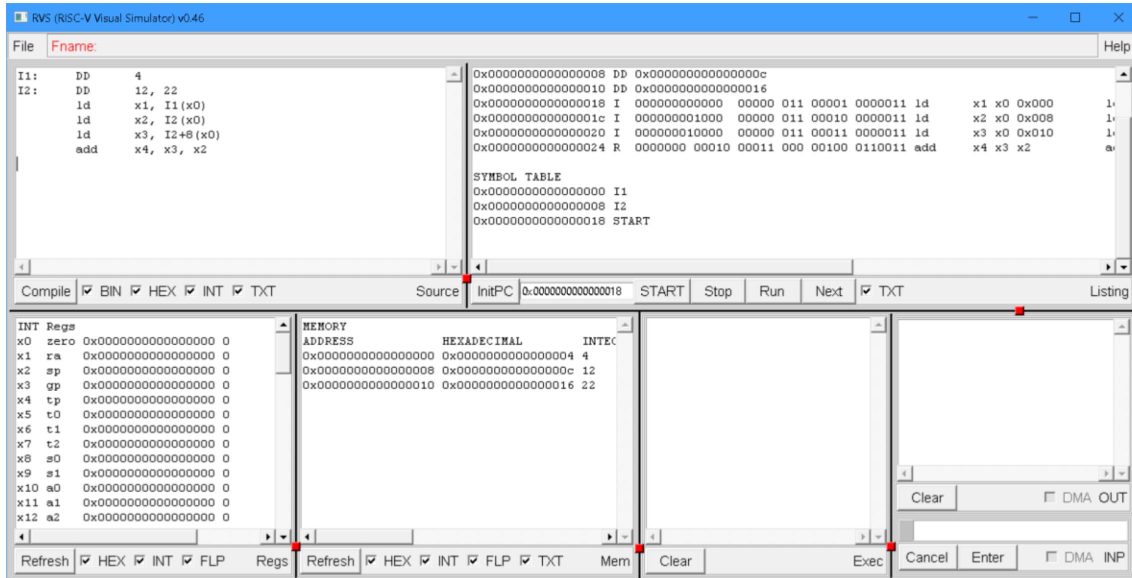
After entering the above code, the **Source** window should appear as follows:



Compile the above source by pressing the Compile button. The result in the **Listing** window should be as follows:



The red line in the listing window indicates an error detected in the last line of the entered assembly code. The error message "**rs2 NOT A REGISTER**" indicates that the third register (denoted by `rs2`) in the last instruction (`add x4, x3, z2`) is wrong. Indeed, we have used `z2` as a register name which is not acceptable. Change `z2` to `x2` and recompile the code. The result should be as follows:



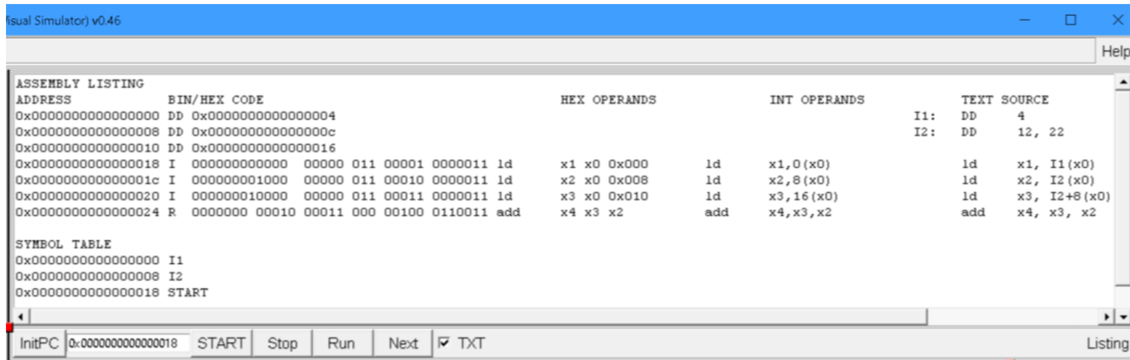
Check the **`SYMBOL TABLE`** in the bottom of the **Listing** window. It shows the values of the `2` labels `I1` and `I2`. The value of `I1` is `0` which is the memory address where the constant `4` was stored. The value of `I2` is `8` which is the memory address where the constant `12` was stored.

The memory addresses and the corresponding stored constants are shown in the **Mem** window above. Note that the third constant `22` is stored at address `16`, right after the second constant `12`.

The first `ld` instruction `ld x1,I1(x0)` computes the address of the constant to load in register `x1` by summing the value of the label `I1` (`0` in our case) with the value of the register `x0` (which is always `0`). The resulting address is thus `0` which refers to the first constant `4`. In a similar way the second `ld` instruction `ld x2,I2(x0)` computes the address `8` of the second constant `12`.

In the third `ld` instruction `ld x3,I2+8(x0)` we specify the address of the third constant `22` by adding `8` to the value of the label `I2`, since there is no label that holds that value.

The check boxes `BIN`, `HEX`, `INT`, and `TXT` next to the **Compile** button in **Source** window control what columns are shown in the **Listing** window after the compilation. Here is a complete view of the **Listing** window when all boxes are checked:



A brief explanation of the corresponding column fields in the ASSEMBLY LISTING is given below.

ADDRESS:

This is the first field from the left and it shows the address of the compiled instruction or data. It shows the memory location address for each source line, for example location `0` holds the integer `4` which is the compiled first source line `I1: DD 4`. Consequently `0xC` (decimal `12`) is stored in location `8`, and `0x16` (decimal `22`) is stored in location `0x10` (decimal `16`) which is the compiled second source line `I2: DD 12,22` . The third source line `ld x1,I1(x0)` is compiled at location `0x18` (decimal `24`). It contains the code of the first compiled machine instruction which is an `ld` (load double) instruction. The address of the first compiled machine instruction is used as a starting address of the program. It is stored in the PC initialization field on the right of the **InitPC** button in the bottom of the **Listing** window.

BIN/HEX CODE (Controlled by the BIN checkbox):

This is the second field from the left and it shows either the binary representation of the compiled machine instruction or the hexadecimal value of the stored data. Consider the memory location 0x18 where the third instruction, namely `ld x1,I1(x0)` is stored. Its binary representation is shown as follows:

```
I  000000000000  00000 011 00001 0000011
```

In the above field `I` indicates the instruction type (immediate). It is followed by the binary representation of the instruction itself. For convenience, the different components of the instruction are separated by spaces.

HEX OPERANDS (Controlled by the HEX checkbox):

This is the third field from the left and it shows the canonical form of the instruction (spaces instead of commas and no parentheses) with all operands in hexadecimal. For the third instruction, namely `ld x1,I1(x0)` this field shows:

```
ld   x1 x0 0x000
```

INT OPERANDS (Controlled by the INT checkbox):

This is the forth field from the left and it shows the standard form of the instruction (with commas and parentheses) with all operands in decimal. For the third instruction, namely `ld x1,I1(x0)` this field shows:

```
ld   x1,0(x0)
```

TEXT SOURCE (Controlled by the TXT checkbox):

This is the fifth field from the left and it shows instruction text as entered in the **Source** window. For the third instruction, namely `ld x1,I1(x0)` this field shows:

```
ld   x1,I1(x0)
```

The inclusion of the above columns in the ASSEMBLY LISTING is controlled by the check-boxes in the bottom of the **Source** window. Only the columns corresponding to the checked boxes will appear in the ASSEMBLY LISTING when the **Compile** button is pressed. Note that in order to change the columns you will have to re-compile the source.

The ASSEMBLY LISTING is followed be a SYMBOL TABLE that contains the values of all the labels used in the program. A brief explanation of the corresponding column fields in the SYMBOL TABLE is given below.

SYMBOL TABLE:

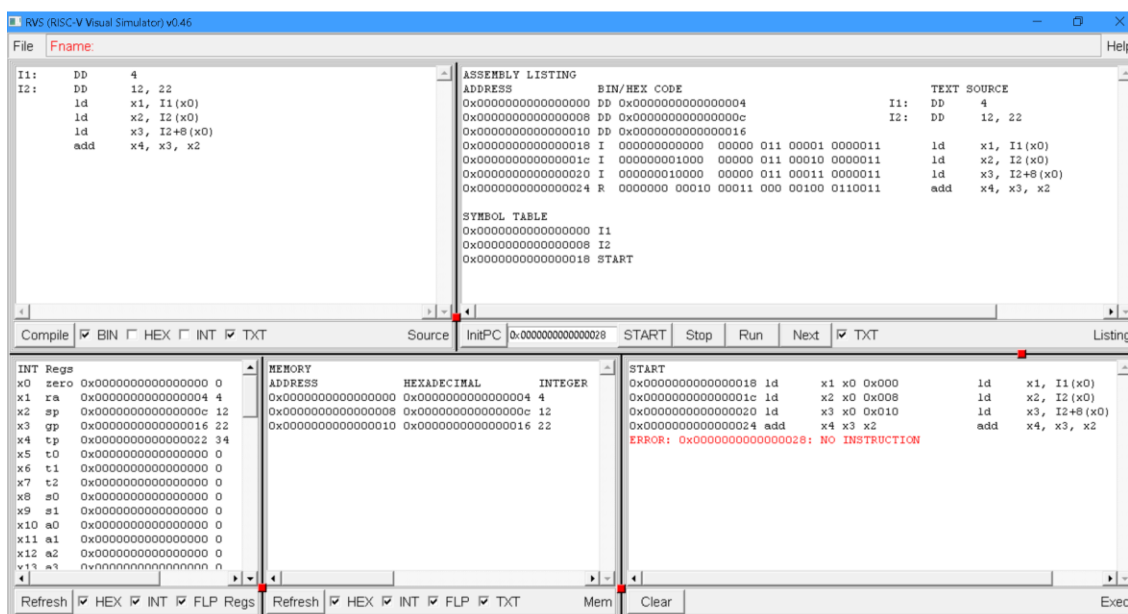
This table has two columns. The labels are shown in the right column and their hexadecimal values are shown in the left column. Note that the rows in the table are shown in the lexicographical order of their corresponding labels. For example, the table for our sample code shows the label `I1` and its value `0x0` in the first row, the label `I2` and its value `0x8` in the second row, and the label START and its value `0x18` in the third row. Note that the label START, which denotes the starting address of the program, was automatically created by the assembler when the first instruction of the code was compiled. The starting address of the program can be explicitly set to another instruction by inserting the START label in front of it.

The following controls are placed in the bottom of the **Listing** window:

- **InitPC** button with an entry field
  The entry field contains the current value of the Program Counter (PC). The PC can be changed by directly typing into this field. Pressing the InitPC button will reset the PC value to the value of the START label (`0x0000000000000018` in our case).
- **START** button
  Pressing this button will reset the PC to the START value and will run the program.
- **Stop** button
  Pressing this button will stop the program at the current instruction (the address of the current instruction will be shown in the PC entry field)
- **Run** button
  Pressing this button will run the program starting from the current instruction (the one pointed by the current value of the PC)
- **Next** button
  Pressing this button will execute the current instruction (the one pointed by the current value of the PC) and then stops. The PC is automatically adjusted to point to the next instruction. Use this button for stepwise execution of your code.
- **TXT** check-box
  This checkbox controls the inclusion of the source code in the **Execution** tracing

4. Execution

The compiled program is executed by clicking on the **START** button in the bottom of the **Listing** window. The resulting RVS window should be as follows:



The execution results are shown in the three bottom row windows as shown on the above snapshot. The information content is as follows.

- The **Regs** window (the leftmost one) shows the state of the registers after the execution. In our case the three register `x1`, `x2`, and `x3` contain the loaded values `4`, `12`, and `22` respectively. The value in register `x4` is `34` which was obtained by summing the values in `x2` and `x3`, namely `12+22`.
- The **Mem** window (the middle one) shows the values in the memory. The memory addresses `0`, `8`, and `16` contain the constants values of `4`, `12`, and `22` respectively. These values were placed there by the assembler when the first two `DD` command in the code were compiled.
- The **Exec** window (the rightmost one) shows the sequence of the executed instructions. The three `ld` instructions were executed in a sequence, followed by the add instruction. The last line in red indicates the end of the program execution. Note that there is no record of executing any instructions corresponding to the first two `DD` commands in the code. Indeed, the `DD` commands only instruct the assembler to place the corresponding constants in the memory at compile time. No machine instructions for execution at runtime are therefore generated when the `DD` commands are compiled.

5. Debugging

When debugging programs, it is often desirable to step through the code and execute it one instruction at a time. Such step-by-step execution is frequently used for code debugging and tracing of intermediate values in registers and in memory. To illustrate this we will use the following simple program that calculates the factorial of 4:

```
;calculates 4!=4*3*2*1=24
      addi    x5, x0, 4
      addi    x6, x0, 1
loop: mul     x6, x6, x5
      addi    x5, x5, -1
      ebreak x0, x0, 0
      blt     x0, x5, loop
```

Enter the above program in the **Source** window of the RVS and press the **Compile** button. Then press the **Next** button once, to step through the code and execute the first instruction. The execution trace in the **Exec** window will appear as follows:

```
NEXT
0x0000000000000000 addi   x5 x0 0x004         addi   x5, x0, 4
```

The value of the register `x5` in the Regs window will appear as follows:

```
x5  t0   0x0000000000000004 4
```

Now press the **Next** button again, to step through the code and execute the second instruction. The following trace will appear in the Exec window:

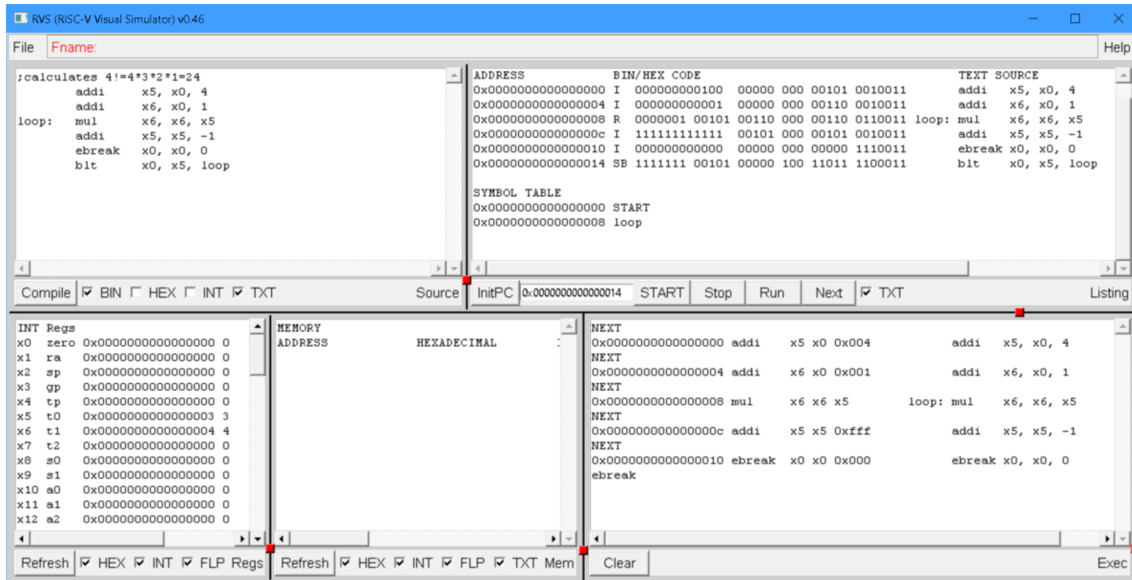
```
NEXT
0x0000000000000004 addi   x6 x0 0x001         addi   x6, x0, 1
```

The value of the register x6 in the Regs window will appear as follows:

```
x6  t1   0x0000000000000001 1
```

By further pressing the **Next** button one can continue stepping through the code and observing how the values in the registers change throughout the first iteration

of the loop. The debugging of the instructions in the loop will be complete at the end of the first loop iteration, so the step-by-step execution could be abolished at this point.



The consequent iterations of the loop can be debugged by halting the execution and checking the respective values just once per iteration. Halting the execution of the program in RVS is implemented by creating a breakpoint through the `ebreak` instruction. In our sample program an `ebreak` instruction has been inserted before the last line of the code. Recompile the code by pressing the **Compile** button. Then press the **START** button to execute the program. The following execution trace will appear in the Exec window:

```
START
0x0000000000000000 addi   x5 x0 0x004          addi   x5, x0, 4
0x0000000000000004 addi   x6 x0 0x001          addi   x6, x0, 1
0x0000000000000008 mul    x6 x6 x5       loop: mul    x6, x6, x5
0x000000000000000c addi   x5 x5 0xfff          addi   x5, x5, -1
0x0000000000000010 ebreak  x0 x0 0x000         ebreak x0, x0, 0
ebreak
```

The values of the registers will be:

```
x5  t0   0x0000000000000003 3
x6  t1   0x0000000000000004 4
```

Note that the initialization and the first iteration of the factorial loop have been completed before the execution has halted at the `ebreak` instruction.

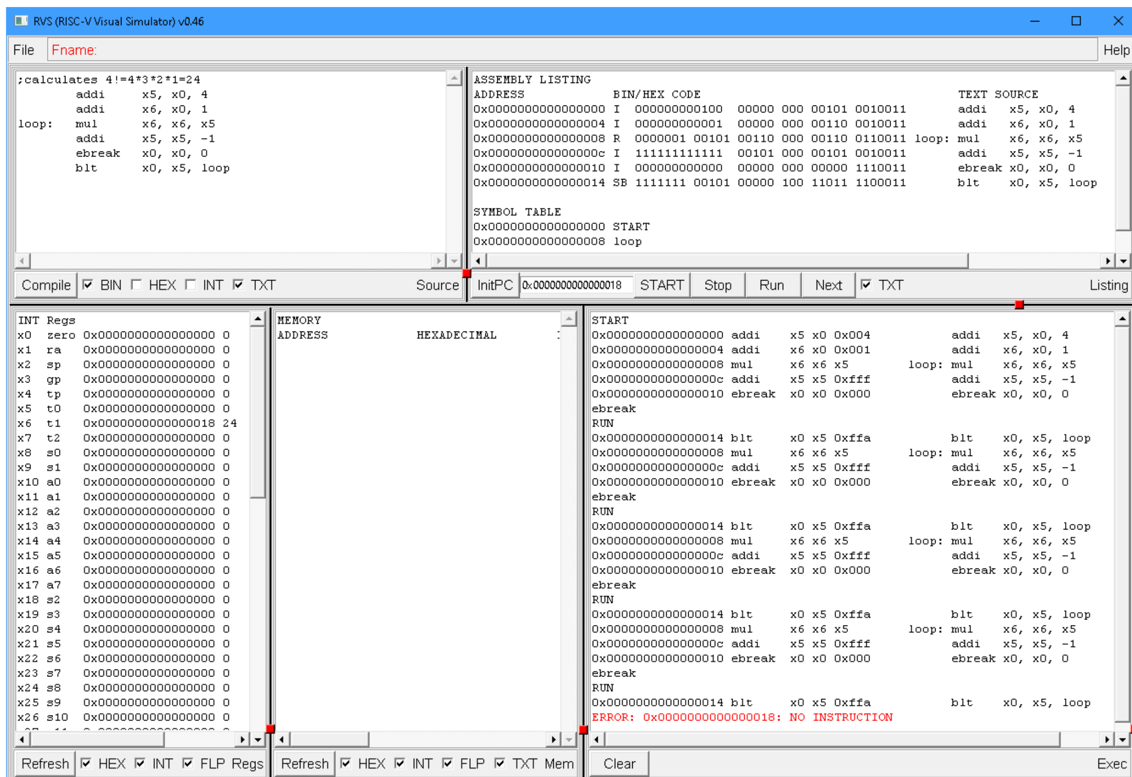
Now press the **Run** button. The following execution trace will appear in the **Exec** window:

```
RUN
0x0000000000000014 blt     x0 x5 0xffa          blt    x0, x5, loop
0x0000000000000008 mul     x6 x6 x5       loop: mul    x6, x6, x5
0x000000000000000c addi    x5 x5 0xfff          addi   x5, x5, -1
0x0000000000000010 ebreak  x0 x0 0x000          ebreak x0, x0, 0
ebreak
```

The values of the registers will be:

```
x5  t0   0x0000000000000002 2
x6  t1   0x000000000000000c 12
```

The second iteration of the factorial loop has been completed before the execution has halted at the `ebreak` instruction.



6. Internals

The RVS execution of RISC-V programs is based on a Virtual memory (VM) model implemented through hash-based associative memory. In this way the executed code can freely refer and use the entire 64-bit addressing space while the allocated physical memory will be confined only to the necessary minimum. With this model there is no need for an operating system to manage the stack and heap memory allocation.

All virtual memory is considered as initialized to 0s so reading from an address without its prior initialization returns 0s without an error while no physical memory is actually allocated. Writing to an address, however, will lead to actual physical memory allocation (if the memory has not been allocated already.) This is also true for writing 0s, e.g. actual physical memory will be allocated and maintained for storing 0s. Although less efficient, this is a design decision that greatly facilitates the tracking of the memory use by the executed code as shown in the **Memory** window.

The RVS provides bit-accurate representation of the supported RISC-V machine instructions although it uses different data structures for simulating the execution process.