

Lab 1. Introduction to RARS

Goals

After this lab session, you will be able to use the RARS tool, write a simple assembly program, simulate the execution of the program, and debug errors (if any). You will also have a general understanding of the operation of a RISC-V processor when it executes instructions.

References

- RISC-V documents, lecture notes.
- The RISC-V Instruction Set Manual: riscv-spec-20191213.pdf

About RARS

- Website: <https://github.com/TheThirdOne/rars>
- RARS version 1.6:
https://github.com/TheThirdOne/rars/releases/download/v1.6/rars1_6.jar

Kick-off

Download and run

1. Download Java Runtime Environment, JRE, to run the RARS tool.
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>
2. Install JRE
3. Download the RARS tool at the following URL:
<https://github.com/TheThirdOne/rars>

The RARS tool can be used immediately without installation. Double-click the file **rars1_6.jar** to run it.

Basics of the IDE programming interface

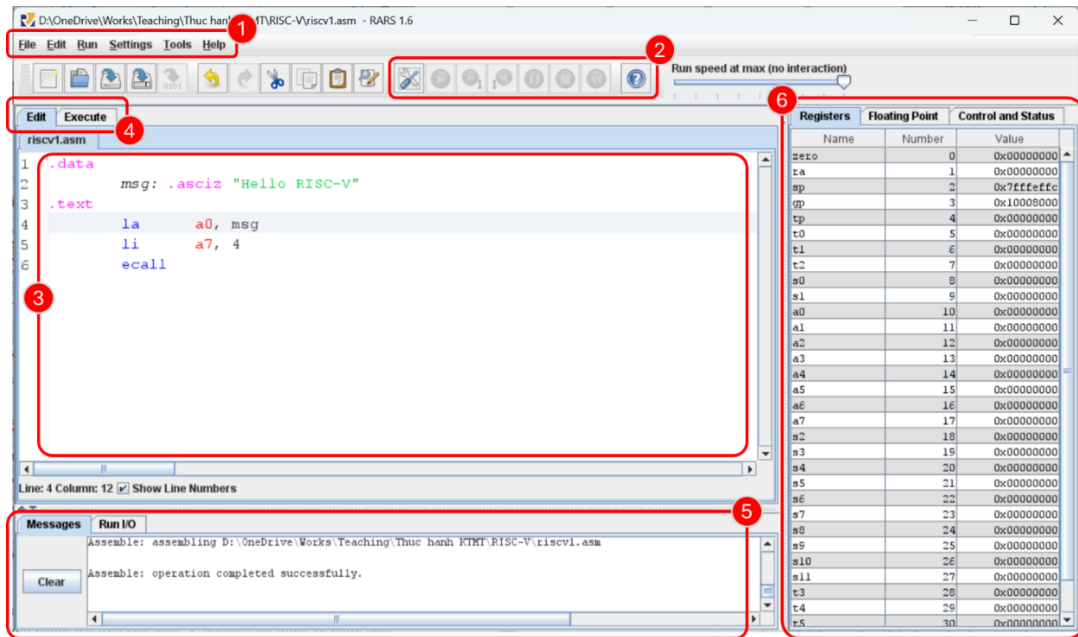


Figure 1. IDE of RARS Tool

1. **Menu:** Most of the items in the menu have corresponding icons.
 - Move the mouse over the icon → A tooltip explaining the corresponding function will appear.

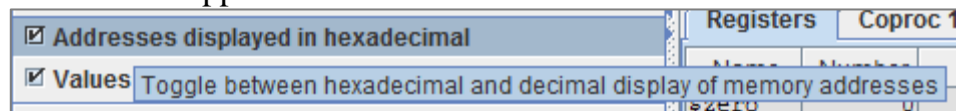


Figure 2. A tooltip explains a function in the menu.

- The items in the menu also have corresponding shortcuts.
2. **Toolbar:**
 - Basic editing features such as copy, paste, open ...
 - Debugging features (in the red rectangle):
 - **Run:** Run the entire program.
 - **Run one step at a time:** Execute one instruction at a time and then pause.
 - **Undo the last step:** Restore the state to the previous instruction.
 - **Pause:** Temporarily pause the running process.
 - **Stop:** End the debugging process.
 - **Reset memory and register:** Restart memory and registers.
 3. **Edit tab:** The RARS tool has a built-in text editor with **syntax highlighting**, making it easier for users to follow the source code. Additionally, when a command is entered but not yet completed, a popup will appear to assist. Go to the Settings / Editor in the menu to change settings related to the editing function.

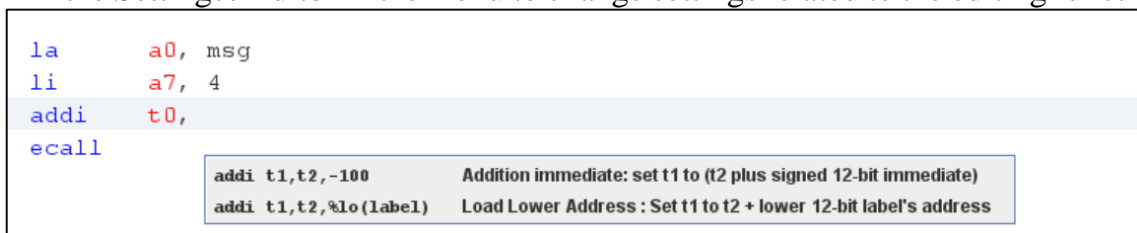


Figure 3. A popup appears to explain the instructions.

4. **Edit/Execute:**

Each source code file in the editing interface has two windows (2 tabs): **Edit** and **Execute**.

- **Edit tab:** Write assembly programs with [syntax highlighting](#).
- **Execute tab:** Compile the assembly program written in the Edit tab into machine code, **run** it, and **debug**.

5. **Message Areas:** There are two windows at the bottom of the IDE interface.

- **Run I/O** is only active when running the program.
 - [It displays results output to the console.](#)
 - [It inputs data into the program via the console.](#)

The RARS tool has an option for all console input data to be displayed again in the message area.

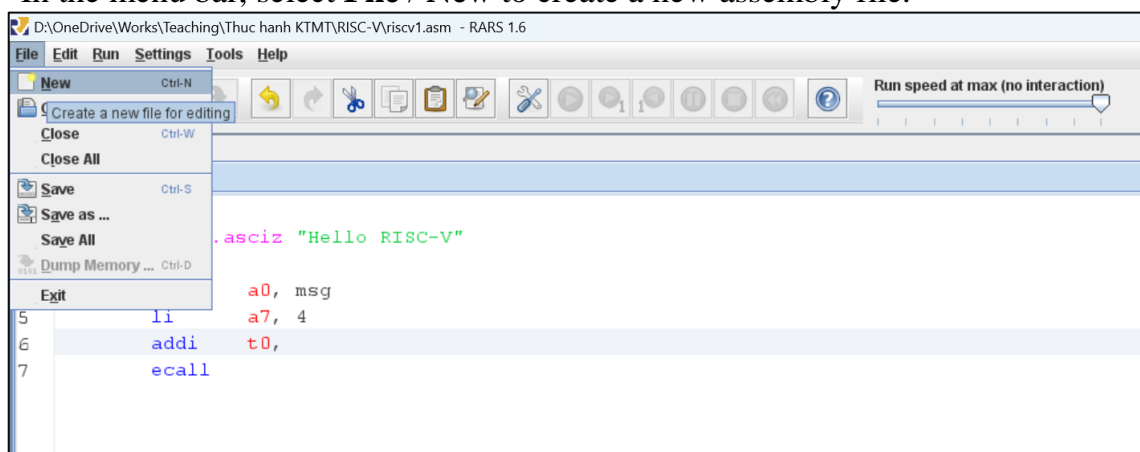
- **Messages** is used to display all other notifications, such as error messages during compilation or while running the program. [Click on an error message to automatically jump to the line causing the error.](#)

6. **Registers:** The table displays the values of the processor's registers, always visible regardless of whether the assembly program is running or not. This table also helps you remember the names and IDs of the registers when writing programs. There are three tabs in this table:

- **General-Purpose Registers** are numbered from 0 to 31, including the *Program Counter* register and can be used in any of the instructions.
- **Floating Point Registers** are used for performing floating-point arithmetic instructions.
- **Control and Status Registers** are used for interrupt handling.

Program and understand the tool with the HelloWorld program

1. Click on the file **rars1_6.jar** to start the program.
2. In the menu bar, select **File / New** to create a new assembly file.



3. The editing window will appear. Start programming.
4. Enter the following code in the editing window.

```
.data                                # Data Segment, declare variables here
```

```
x:      .word    0x01020304    # Declare variable X with initial value
msg:    .asciz   "School of Information and Communications Technology"
.text   # Text Segment, instructions, program by asm language
la      a0, msg                # Get the address of the variable msg, store into a0
li      a7, 4                  # Set register a7 = 4
ecall   # Invoke system procedure to print string msg

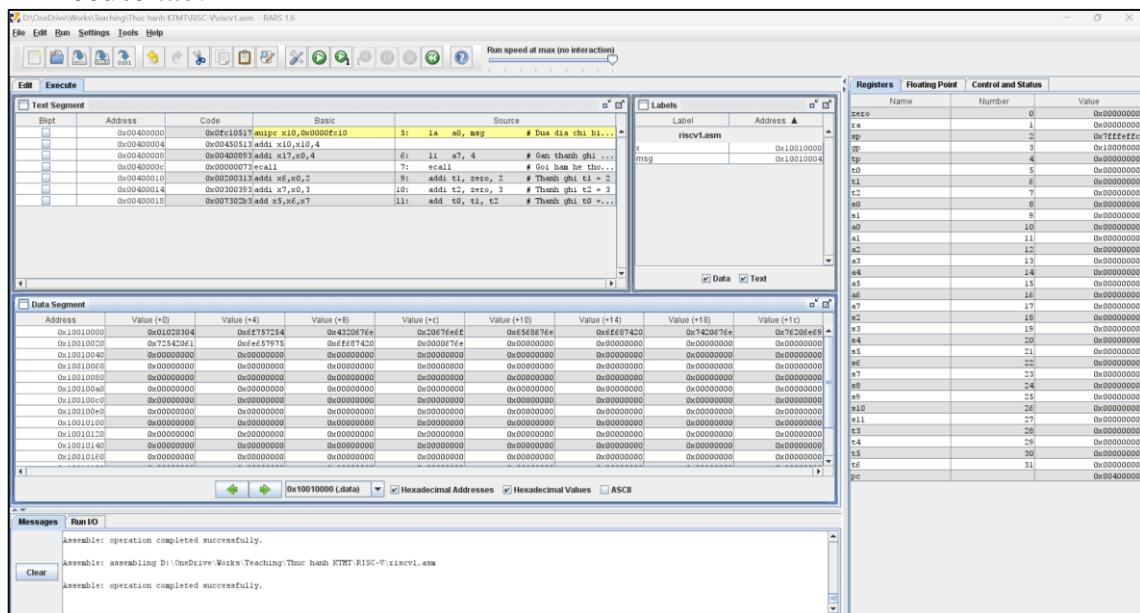
addi    t1, zero, 2            # Register t1 = 2
addi    t2, zero, 3            # Register t2 = 3
add     t0, t1, t2              # Register t0 = t1 + t2
```

- To compile the assembly program into machine code, do one of the following:
 - Go to **Run / Assemble** in the menu.

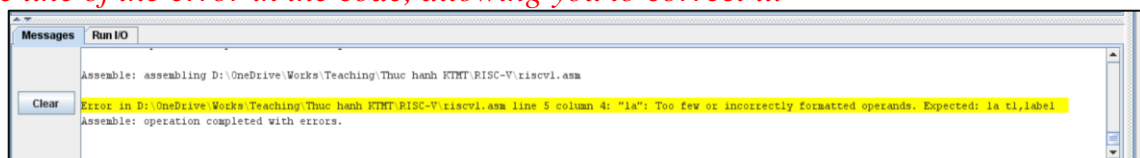


- Click the icon in the menu bar.
- Press the **F3** shortcut key.

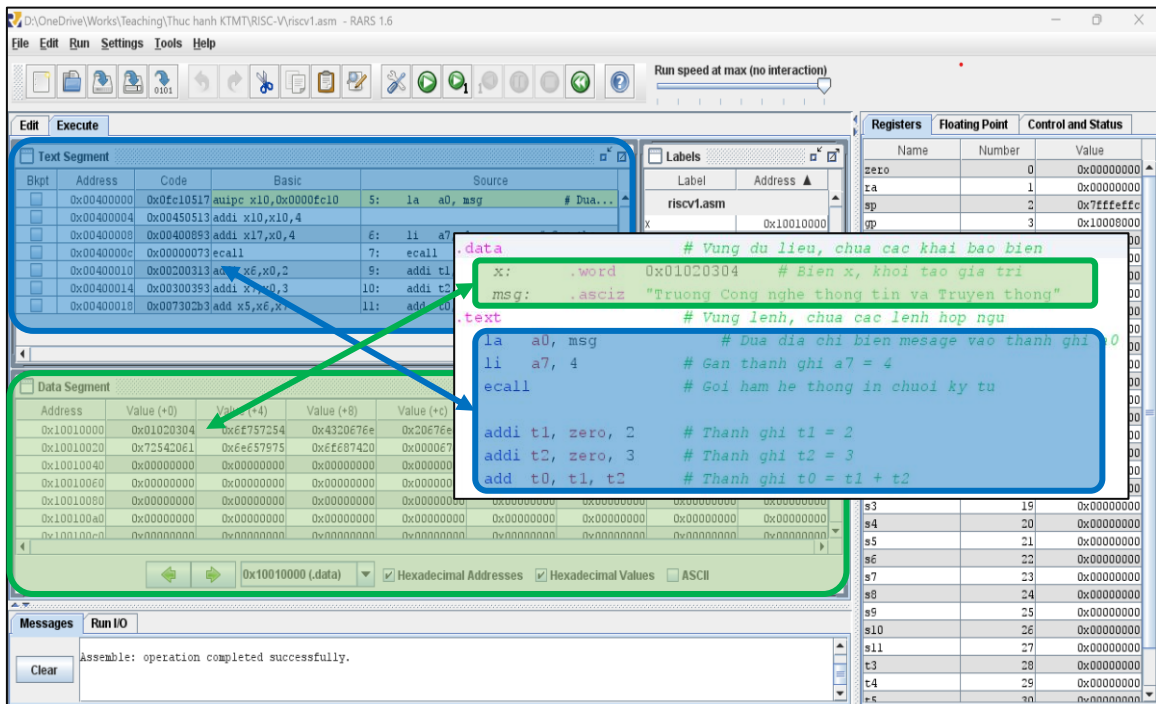
- If the assembly code is correct, the RARS tool will switch from the **Edit** tab to the **Execute** tab.



Note: If the assembly code has an error, the Messages window will display the details of the error. Click on the error message line, and the editor will automatically jump to the line of the error in the code, allowing you to correct it.



- In the **Execute** tab, there are two main windows: **Text Segment** and **Data Segment**.

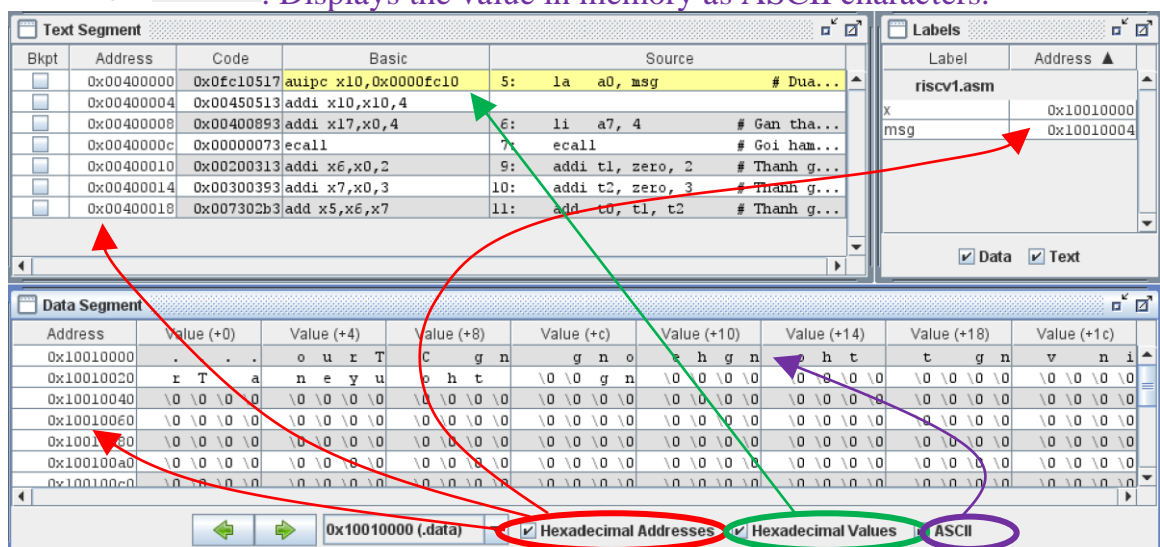


- **Text Segment** is the memory space that contains the assembly code. In the assembly source code, any code written after the `.text` directive belongs to the Text Segment.
- **Data Segment** is the memory space that contains variables. In the assembly source code, any code written after the `.data` directive belongs to the Data Segment.

Note: For some reason, if you declare a variable after the `.text` directive or vice versa, the compiler will either report an error or ignore the incorrect declaration.


8. In the **Execute** tab, use the checkboxes below to change the data display format for easier observation.

- ☐ **Hexadecimal Addresses** : Displays the address in the hexadecimal format.
- ☐ **Hexadecimal Values** : Displays the register value in the hexadecimal format.
- ☐ **ASCII** : Displays the value in memory as ASCII characters.



9. In the **Execute** tab, within the **Text Segment** window, the table has five columns:

Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x0fc10517	auipc x10,0x0000fc10	5: la a0, msg # Dua...
<input type="checkbox"/>	0x00400004	0x00450513	addi x10,x10,4	
<input type="checkbox"/>	0x00400008	0x00400893	addi x17,x0,4	6: li a7, 4 # Gan tha...
<input type="checkbox"/>	0x0040000c	0x00000073	ecall	7: ecall # Goi ham...
<input type="checkbox"/>	0x00400010	0x00200313	addi x6,x0,2	9: addi t1, zero, 2 # Thanh g...
<input type="checkbox"/>	0x00400014	0x00300393	addi x7,x0,3	10: addi t2, zero, 3 # Thanh g...
<input type="checkbox"/>	0x00400018	0x007302b3	add x5,x6,x7	11: add t0, t1, t2 # Thanh g...

- **Bkpt**: Breakpoints, the stopping points when running the entire program using the  button.
- **Address**: The address of the instruction in the integer format (*see more instructions in the Label window*).
- **Code**: The instruction in the machine code format.
- **Basic**: The instruction in assembly language, as specified in the instruction set. Here, all labels, mnemonics, etc., have been converted into constants.
- **Source**: The instruction in assembly language with additional macros, labels, etc., to help write a program faster and more understandable. For example:
 - The **la** instruction in the Source column is a pseudo-instruction, not part of the instruction set, and is translated into two corresponding instructions, **auipc** and **addi** in the Basic column.
 - The **msg** label in the **la a0, msg** instruction in the Source column is replaced by parameters for the **auipc** and **addi** instructions.

10. In the **Execute** tab, within the **Data Segment** window, the table has nine columns:

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x01020304	0x6f757254	0x4320676e	0x20676e6f	0x6568676e	0x6f687420	0x7420676e	0x76206e69
0x10010020	0x72542061	0x6e657975	0x6f687420	0x0000676e	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000



```

.data                                     # Vung du lieu, chua cac khai bao bien
x:                                     # Bien x, khoi tao gia tri
    .word 0x01020304
msg:                                  .asciz "Truong Cong nghe thong tin va Truyen thong"
  
```


Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000
0x10010020	r	T	a	n	e	y	u	.
0x10010040	\0	\0	\0	\0	\0	\0	\0	\0
0x10010060	\0	\0	\0	\0	\0	\0	\0	\0
0x10010080	\0	\0	\0	\0	\0	\0	\0	\0
0x100100a0	\0	\0	\0	\0	\0	\0	\0	\0
0x100100c0	\0	\0	\0	\0	\0	\0	\0	\0

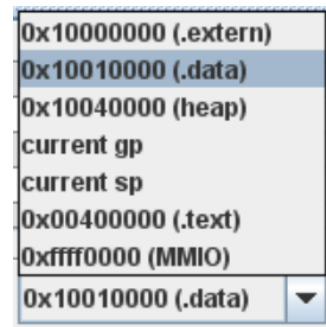
- **Address**: The address of the data or variable in the integer format. The value of each row increases by 32 in decimal or 0x20 in hexadecimal because each row presents 32 bytes at consecutive addresses.
- **Value columns**: Each column contains 4 bytes, and there are 8 columns, corresponding to 32 consecutive bytes.

In the image above, the value of the variable $x = 0x01020304$ is displayed correctly in the Data Segment with the number format ☐ ASCII, and the value of the string “Truong Cong nghe thong tin va Truyen thong” is displayed correctly with the ASCII format ☒ ASCII. *Note that storing strings in memory in the little-endian format is due to the way the software syscall function is programmed, not a requirement of the MIPS processor. As can be seen, the RARS tool stores strings in the big-endian format.*

Click the pair of buttons  to move to the neighboring address region.

Click on the ComboBox to navigate to the memory region containing the specified data type:

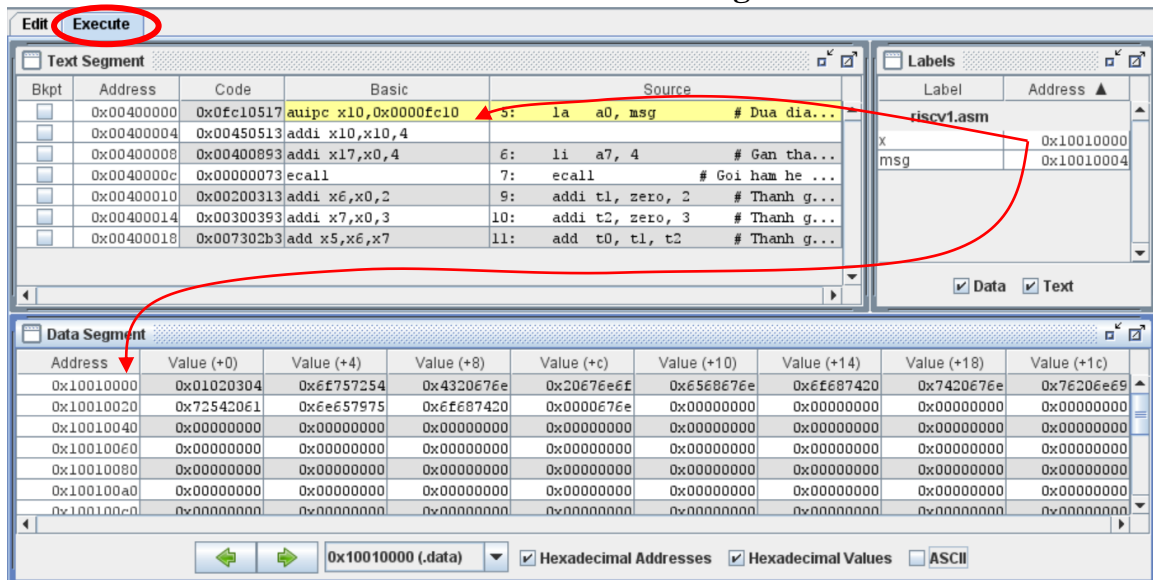
- **.data**: Data region
- **.text**: Instruction region
- **sp**: Stack region



11. The **Label** window displays the label names and the corresponding address constants when compiled into machine code. The **Label** window does not display automatically. In the menu, go to the Settings / Show Labels Window to toggle the display of the **Label** window.

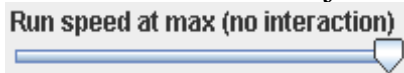
In the following image, important information can be seen:

- In the **Label** window:
 - o **x** is just a mnemonic, and it will be converted into 0x10010000 (constant).
 - o **msg** is also just a mnemonic, and it will be converted into 0x10010004 (constant).
 - o Double-clicking on a variable name will automatically jump to the corresponding position in the **Data Segment** window.
- In the **Text Segment** window:
 - o In the **la a0, msg** assignment instruction, the mnemonic **msg** has been converted into 0x10010004 through the **auipc** and **addi** instruction pair.
- In the **Data Segment** window:
 - o To monitor the value of the variable **x**, open the **Data Segment** at 0x10010000 to see the value of **x**.
 - o To monitor the value of the variable **msg**, open the **Data Segment** at 0x10010004 to see the value of **msg**.







Running the Emulator

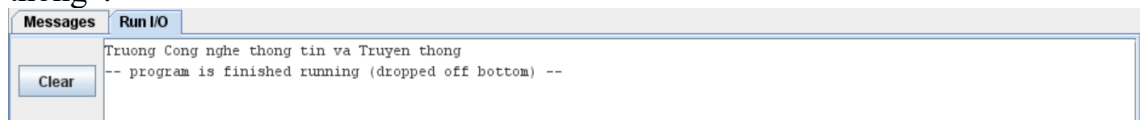
1. Continue running the *Hello World* program mentioned above.
2. Use the slider bar to adjust the execution speed of the instructions.



By default, the execution speed is set to maximum, and at this speed, it's difficult to intervene much in the operation of the instructions and control them. However, you can move the slider bar to around one or two instructions/second for easier observation.

3. In the **Execute** tab, choose a way to run the program:

- Click the **Run** icon  to execute the entire program. When using **Run**, you can observe the line highlighted in yellow, indicating where the assembly program is being processed. Simultaneously, you can observe the data changes in the **Data Segment** window and the Registers window.
- Click the **Reset** icon  to restart the emulator to its initial state. All memory blocks and registers are reset to 0.
- Click the **Run one step** icon  to execute just one instruction and then wait to click on the icon again to execute the next instruction.
- Click **Run one step backwards** icon  to restore the state and go back to the previously executed instruction.
- After running all the instructions of the *Hello World* program, you will see the **Run I/O** window display the string “Truong Cong nghe thong tin va Truyen thong”.



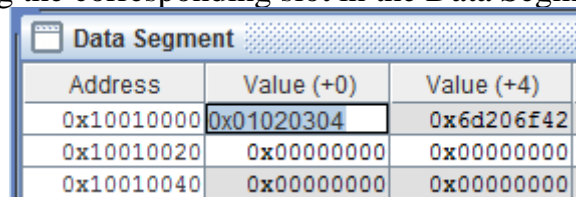
Emulation & Debugging: Observe Changes in Variables

During emulation, execute one instruction at a time with the **Run one step** function. At each instruction, observe the value changes in the **Data Segment** and **Registers** windows, and understand the meaning of those changes.

Emulation & Debugging: Change Variable Values at Runtime

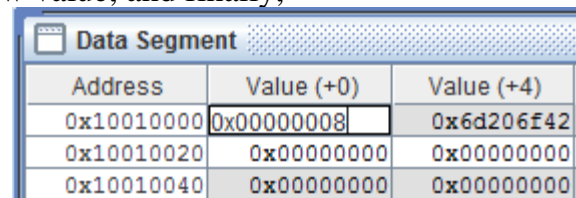
During emulation, you can change the value of any memory block by:

- Double-clicking the corresponding slot in the Data Segment, and then



Data Segment		
Address	Value (+0)	Value (+4)
0x10010000	0x01020304	0x6d206f42
0x10010020	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000

- Entering the new value, and finally,



Data Segment		
Address	Value (+0)	Value (+4)
0x10010000	0x00000008	0x6d206f42
0x10010020	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000

- Continuing running the program.

Emulation & Debugging: Change Register Values at Runtime

The method is similar to changing variable values but applied to the **Registers** window.

Registers	Floating Point	Control and Status
Name	Number	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x7ffffc
gp	3	0x10008000
tp	4	0x00000000
t0	5	0x00000005
t1	6	0x00000002
t2	7	0x00000003
s0	8	0x00000000
s1	9	0x00000000
a0	10	0x10010004
a1	11	0x00000000
a2	12	0x00000000

Consulting Help

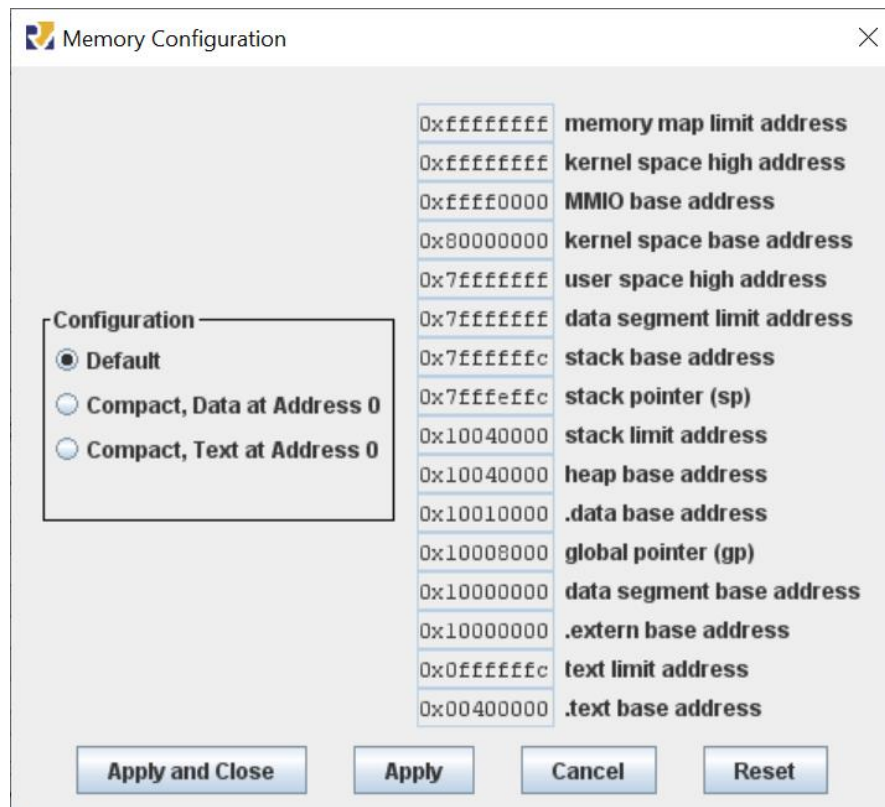


Click the  button to view explanations of RISC-V instructions, pseudo-instructions, directives, and system service functions.

Memory Address Constants

In the menu, Choose **Settings / Memory Configuration...**

The **Memory Configuration** window contains a table defining the memory address constants used by the RARS tool. According to this table, machine code always starts at address 0x00400000, and data always starts at address 0x10000000.



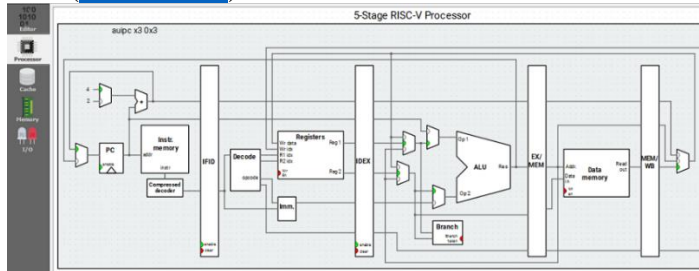
Lab 1 (extended). Introduction to Other Tools

RISC-V is a well-known architecture used by many CPU/MCU lines, including:

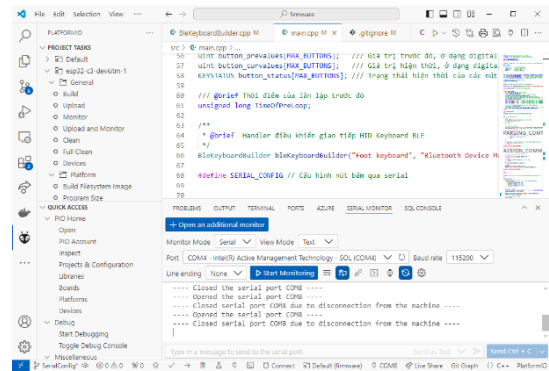
- ESP32 C3 and C6 series by Espressif ([See details](#))
- CH32 by WCH ([See details](#))
- Allwinner D1 with MangoPi board ([See details](#))

There are many other simulation and emulation tools, such as:

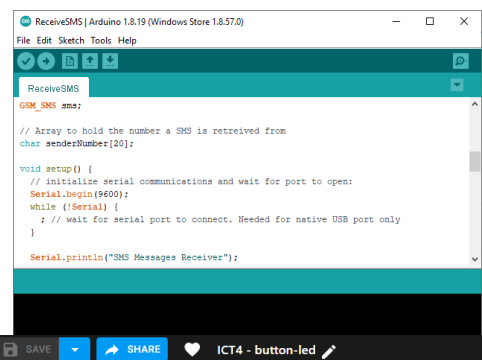
1. **Ripes**, used to simulate code written in C or assembly language on a RISC-V processor with the RV32I[M][C] instruction set. Available on both Web and PC. ([See details](#))



2. **Visual Studio Code**, combined with the **PlatformIO** extension, used for programming in C and assembly on real RISC-V processors, such as the ESP32-C3. This tool supports direct debugging on RISC-V processors. No simulation. Available on PC.



3. **ArduinoIDE**, used for programming in C and assembly on real RISC-V processors, such as the ESP32-C3. This tool does not support direct debugging on RISC-V processors. No simulation. Available on PC.



4. **Wowki**, a visual tool for designing circuit boards with many sensors and actuators. It is helpful for simulating IoT designs. Available on Web. ([See details](#))

