

# Lab 2. Instruction Set, Basic Instructions, Compiler Directives

## Goals

After this lab session, you will understand modules of computer system, how it works by debugging simple instructions of a RISC-V Processor. You also know and use basic assembly instructions and find out the nature of CPU Architecture, exploit debug tools to verify knowledge of Computer Architecture and Instruction Set. Remember some common Compiler Directives which are used to guide RARS complete source code correctly.

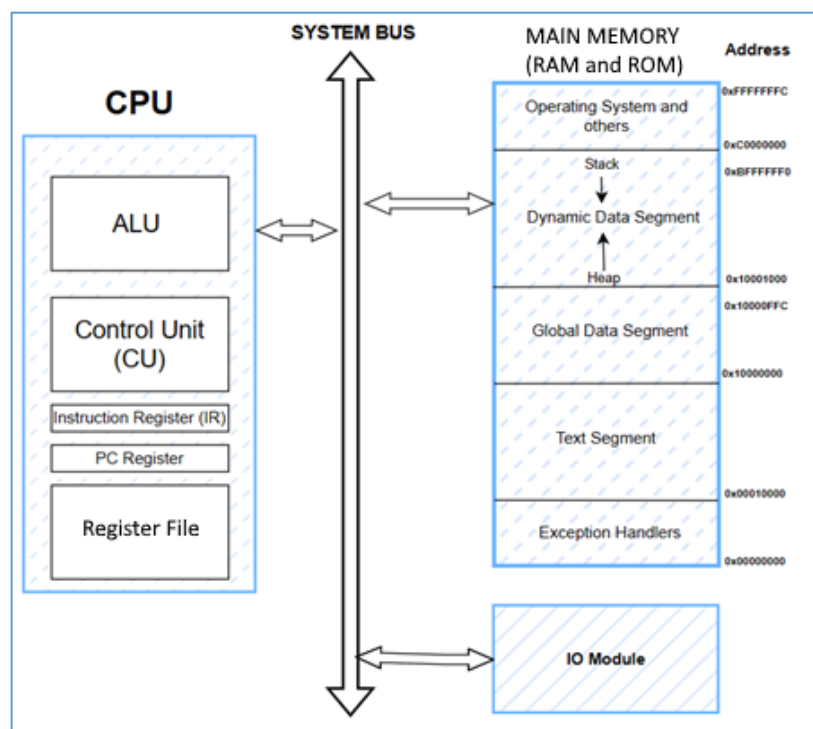
## References

- RISC-V documents, lecture notes.
- The RISC-V Instruction Set Manual: [riscv-spec-20191213.pdf](https://riscv.org/specifications/instruction-set-manual/)

## Home Assignments and Assignments

### Home Assignment 1

Survey and try to have a glance of computer architecture: CPU, Memory, IO Modules and System Interconnection (Bus); Programming Model; Dataflow; Instruction Set Architecture;



A set of registers, called Register File, act as built-in variables inside the CPU. Developers use registers as command variables, pointer variables point to difference locations in the main memory such as Operating System, Text Segment, Data Segment, Stack...

## Home Assignment 2

Read more about RISC-V architecture, remember fundamental knowledge as below:

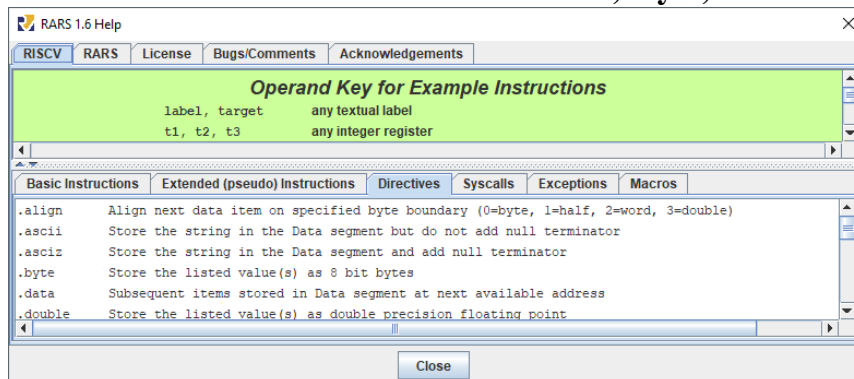
- Names and meanings of 32 registers
- Dedicated registers PC, IR
- The simplest Instruction Set called RV32I. its extension with letters more
  - + letter **M**: supports Multiplication
  - + letter **C**: support compacted instruction with the length of 16-bit
 For example, RV32IMC, RV32IM, etc.
- The RV32I has about 40 instructions which are classified into 6 groups, called instruction formats: R, I, S, B, U, J.

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0			
funct7				rs2			rs1		funct3		rd			opcode		R-type	
imm[11:0]						rs1		funct3		rd			opcode		I-type		
imm[11:5]				rs2			rs1		funct3		imm[4:0]			opcode		S-type	
imm[12]		imm[10:5]			rs2			rs1		funct3		imm[4:1]		imm[11]		opcode	B-type
imm[31:12]										rd			opcode			U-type	
imm[20]		imm[10:1]			imm[11]		imm[19:12]			rd			opcode			J-type	

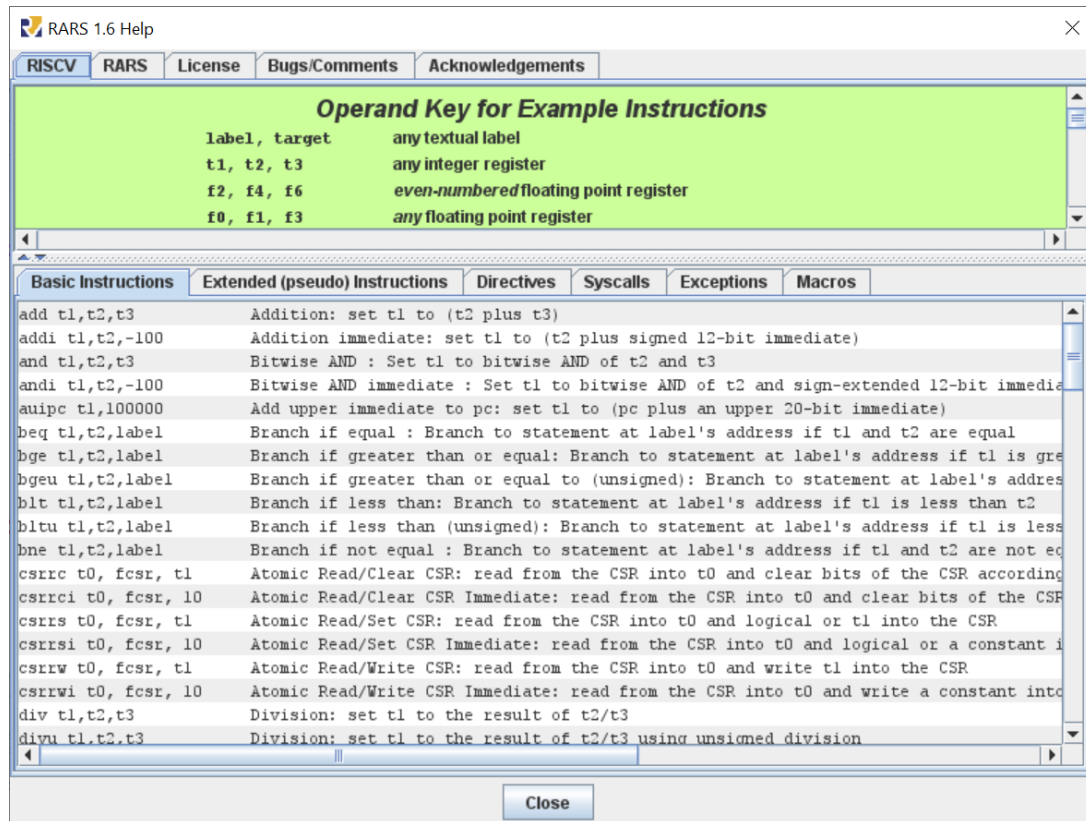
## Home Assignment 3

On the menu, select **Help** / click **Help**.

- Click on tab **Directives**. Understand **.asciiz**, **.byte**, **.word**, **.data**, **.text**



- Click on tab **Basic Instructions**. Search information about instructions **add**, **addi**, **lui**, **mul**, **lw**
- Click on tab **Extended (pseudo) Instructions**. Search information about instructions **li**, **la**



## Assignment 1: Assign 12-bit integer numbers / small integer

**Support:** As a RISC architecture, RV32I instruction set does not have an instruction to assign values directly to registers, but can use the addition instruction to assign. For example, to assign the value `0x123` to register `s0`, use the addition instruction **`addi s0, zero, 0x123`**.

On the other hand, instruction **`addi`** format is I type. I format save 12-bits to store 12-bit signed integer (imm[11:0]). As a result, **`addi`** just assign a small integer in the range of 12-bits (from -2048 to 2047).

Copy the following program into the RARS emulator:

```
# Laboratory Exercise 2, Assignment 1
.text
    addi s0, zero, 0x512    # s0 = 0 + 0x512; I-type: just store a constant
                             # with 12-bit length
    add  s0, x0, zero       # s0 = 0 + 0 ; R-type:
```

### Requirements:

- Use debug tool. Run step by step.
- Monitor the Registers window:
  - o Register **`s0`**.
  - o Register **`PC`**.

- Learn more about instructions **lb**, **sb**.
- In Text Segment, determine the instruction formats of the machine codes.
- Modified the instruction **addi** as bellow. Explain the result.

```
addi s0, zero, 0x20232024
```

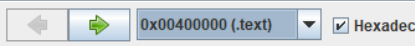
## Assignment 2: Assign 32-bit integer

**Support:** To assign a 32-bit number to a 32-bit register, you let instruction *load upper immediate* (**lui**), and instruction **addi** more. **lui** format is U-type which contains a literal number of 20-bit. It loads a 20-bit constant into the 20 most significant bits of the destination register. Combined with **addi**, **addi** loads a 12-bit constant into the 12 most significant bits of the destination register. So you have full 32 bits.

Copy the following program into the RARS emulator:

```
# Laboratory Exercise 2, Assignment 2
# Load 0x20232024 to s0 register
.text
    lui s0, 0x20232    # s0 = 0xABCDE000
    addi s0, s0, 0x024 # s0 = s0 + 0x123
```

### Requirements:

- Use debug tool. Run step by step.
- Monitor the Registers window:
  - o Register **s0**.
  - o Register **PC**.
- In Text Segment, determine the instruction formats of the machine codes.
- Modified the instruction **addi** as bellow. Explain the result.
- In Data Segment, click to Combo Box, select **.text** to show values of memory inside instruction area (**.text**).
  - o Compare data in Data Segment with machine codes in Text Segment.

### Note:

- In RISC-V, the constant (immediate value) is always 2's complemented, should be extend to 32-bit 2's complement numbers, to fit the length of register.

```
# Laboratory Exercise 2, Assignment 2
# IN HIGH LEVEL LANGUAGE
# int a = 0xFEEDB987;
# IN ASSEMBLY LANGUAGE
.text
    lui s0, 0xFEEDC    # s0 = 0xFEEDC000
    addi s0, s0, 0xFFFFF987 # s0 = 0xFEEDB987
```

### Assignment 3: new assignment instructions

**Support:** As a Reduced Instruction Set Computer architecture, RISC-V was optimized to be simpler with fewer instructions. Consequently, developers must write code longer. To compensate, the compilers support some Extended/Pseudo Instructions, which are not part of the RISC-V instruction set but easier for developers to program. Whenever compiled to machine code, each pseudo instruction could be 1 or more real instructions.

Copy the following program into the RARS emulator:

```
# Laboratory Exercise 2, Assignment 3
.text
    li s0, 0x20232024
    li s0, 0x20
```

#### Requirements:

- Compile, observe and compare the commands in the Source column and the Basic column in the Text Segment window. Explain the results.

### Assignment 4: Calculate the expression $2x + y = ?$

Copy the following program into the RARS emulator:

```
# Laboratory Exercise 2, Assignment 4
.text
    # Assign X,Y into t1,t2 register
    addi t1, zero, 5      # X = t1 = ?
    addi t2, zero, -1     # Y = t2 = ?

    # Expression Z = 2X + Y
    add s0, t1, t1        # s0 = t1 + t1 = X + X = 2X
    add s0, s0, t2        # s0 = s0 + t2 = 2X + Y
```

#### Requirements:

- Use debug tool. Run step by step.
- Monitor the Registers window:
  - o Register **t1**, **t2**, **s0**,
  - o Value of **s0** is correct?
- In Text Segment window, obtain machine codes of **addi** (I-type) and **add**(R-type) instructions.
- Try to compile **addi** (I-type) and **add**(R-type) to machine code manually, and compare.

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0	
funct7					rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]						rs1		funct3		rd		opcode		I-type	

## Assignment 5: Multiplication

**Support:** Multiplication is quite different from other mathematical instructions, because when multiplying two 32-bit numbers, the result is a 64-bit number. The RISC-V architecture provides different instructions for performing multiplication, which can write the result as 32-bit or 64-bit, depending on the instruction. These instructions are not part of the RV32I basic architecture, but are part of the RV32M extension (RISC-V multiply/divided extension).

The first multiplication type:

```
mul    rd, rs1, rs2 # rd = 32 least signification bits of rs1 * rs2
```

The second multiplication type:

```
mul     rd, rs1, rs2 # rd = 32 least signification bits of rs1 * rs2
mulh    rd, rs1, rs2 # rd = 32 msb of rs1 * rs2 (both rs1, rs2 are signed)
mulsu   rd, rs1, rs2 # rd = 32 msb of rs1 * rs2 (1 signed , 1 unsigned)
mulhu   rd, rs1, rs2 # rd = 32 msb of rs1 * rs2 (both rs1, rs2 are unsigned)
```

Copy the following program into the RARS emulator:

```
# Laboratory Exercise 2, Assignment 5
.text
    # Assign X, Y into t1, t2 register
    addi t1, zero, 4      # X = t1 =?
    addi t2, zero, 5      # Y = t2 =?

    # Expression Z = X * Y
    mul s1, t1, t2        # s1 just stores the low 32 bits of the result
```

### Requirements:

- Use debug tool. Run step by step.
- Monitor the Registers window:
- Verify results of multiplication instructions.
- Try division instructions

## Assignment 6: Declare and access variables

**Support:** Compiler directives are not machine codes, but supply more information for the compiler to compile assembly code more accurately.

From the perspective of CPU, both instructions and data are binary numbers. cannot.

CPU cannot distinguish between instructions and data. In main memory, is a hexadecimal number 0xF0028293 an instruction or a variable? Let us verify.

- Play a role of CPU, a hardware:
  - o If the Program Counter register, PC, is pointing to that number, that number should be an instruction which is `addi x5, x5, -256`.
  - o If PC isn't point to that number, the number should be a variable. You can modify values, read/write it.
- Play a role of a compiler, a software:
  - o If that number is declared after the directive **.data**, it is a variable.
  - o If that number is declared after the directive **.text**, it is an instruction and should be disassembled to `addi x5, x5, -256`

The directive **.data** and **.text** works as bookmarks, locate the start address of a certain memory area in RAM, where the compiler will set the first variable or the first instruction. This starting point is purely a convention for controlling resources, so each CPU, each operating system, or compiler can set different starting points.

The example below shows 2 directives **.data** and **.text** for allocating and initializing global variables, defining constants, variables loaded into the data segment, and the code loaded into the text segment. (Understand how the program works).

Copy the following program into the RARS emulator:

```
# Laboratory Exercise 2, Assignment 6
.data                                # Declare variables
X: .word 5                          # Variable X, word type (4 bytes), initial value = 5
Y: .word -1                         # Variable Y, word type (4 bytes), initial value = -1
Z: .word 0                          # Variable Z, word type (4 bytes), initial value = 0

.text                                # Declare instructions
# Fetch values of variables X and Y to registers
la t5, X                            # Get the literal address of X in Data Segment
la t6, Y                            # Get the literal address of Y
lw t1, 0(t5)                        # t1 = X
lw t2, 0(t6)                        # t2 = Y

# Calculate Z = 2X + Y via registers respectively
add s0, t1, t1
add s0, s0, t2

# Store values from registers to main memory (RAM)
la t4, Z                            # Get Z's address
sw s0, 0(t4)                        # Store Z's value to main memory
```

### Requirements:

- In Labels window, see the addresses of X, Y, Z in main memory.
  - o Double click to labels X, Y, Z in the Labels windows to highlight their locations in Data Segment. Read values and compare with initial values in the assembly program.
- Compile and monitor instructions in the Text Segment window.
  - o How is Instruction **la**, load address, compiled (code column, basic column, address column). Explain the way **la** works. (keep an eye on the register **pc**, address of label X, Y).
- Use debug tool. Run step by step.
- Monitor the Registers window:
  - o Values of registers
  - o **lw** and **sw**.
- Learn more about instructions **lb**, **sb**.
- **Remember the processing rules:**
  - o Load as much as possible variables from main memory into registers by instructions **la**, **lw**.
  - o Calculate with registers.
  - o Store values from register back to main memory by instructions **la**, **sw**.

## Assignment 7: Declare variables or instructions at specified addresses

**Support:** you could declare directives **.data**, **.text** with syntax below

```
.data literal_address  
.text literal_address
```

The *literal address* is the start address of the memory area. The compiler will allocate variables or instructions from this address onwards.

### Consequences:

- Value of variables could be changed, but its address is fixed (*Ignore virtualization techniques or operating system*).
- The address of an instruction is fixed.

*Note:*

- If you can **find out the address of a variable** of a target software, you can develop another software to illegally access that variable and change its value. That is hacking.
- If you can **find the address of a instruction** of a target software, you can develop another software to replace that **instruction** by a jump instruction to make CPU move to your codes. That is a computer virus.

```
# Request the compiler to allocate variable at the address of 0x10011234  
.data 0x10011234  
x: .word
```



```
# Request the compiler to allocate variable at the address of 0x10014320
.data 0x10014320
y: .word

# # Request the compiler to allocate instruction at the address of 0x00408000
.text 0x00408000
addi x1, zero, 2
```

---

## ***Conclusion***