Computer Organization

Lab2 RISC-V Assembly language

Data Details

Topics

- RISC-V introduction
- Rars introduction
- Data Processing Details
  - Data transfer: load & store
  - Address alignment
  - Data interpretation
- Practice

RISC-V introduction

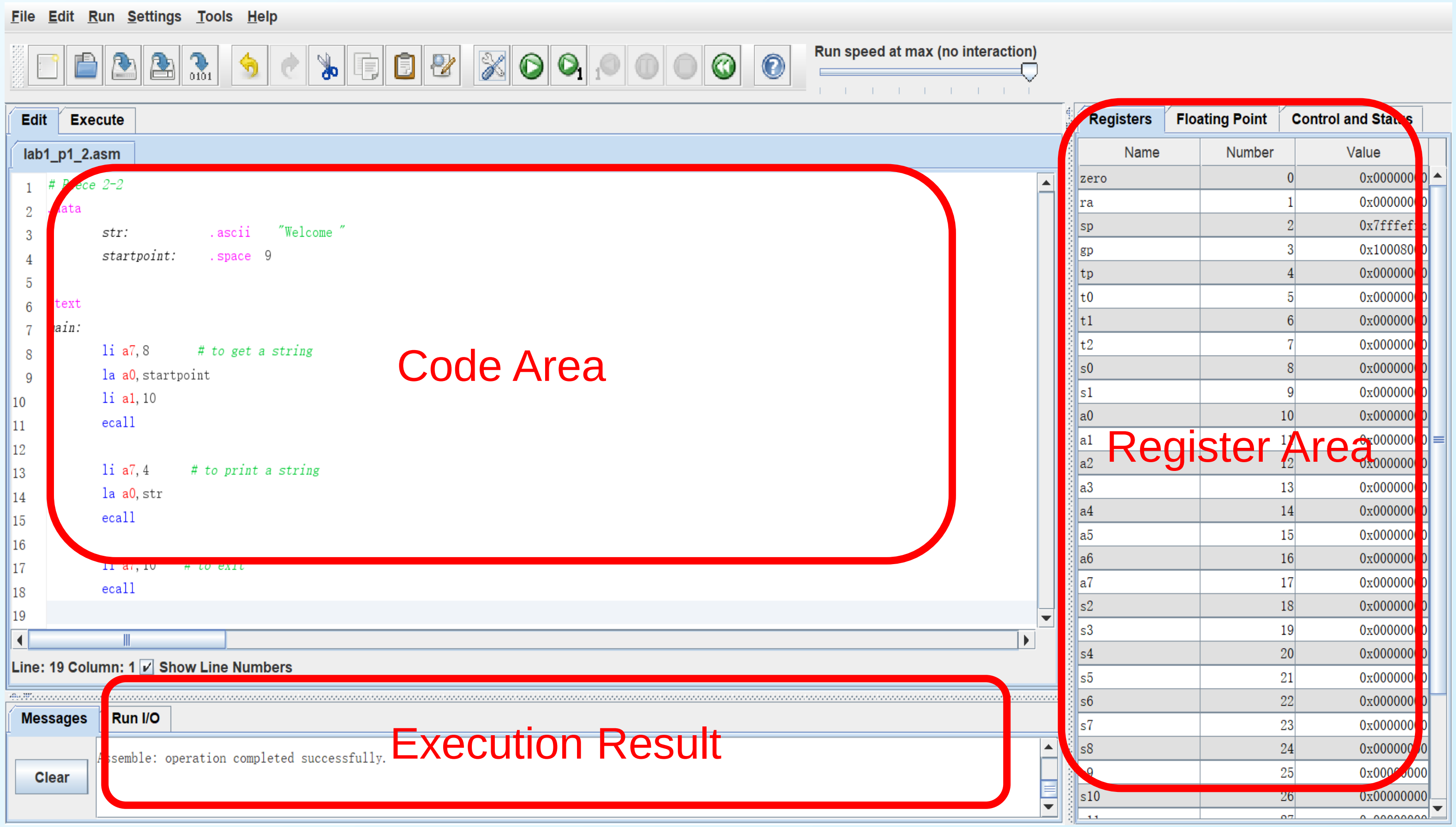
- New open-source, license-free ISA spec.
- Appropriate for all levels of computing system, from microcontrollers to supercomputers.
- 32-bit, 64-bit, and 128-bit variants. (we use 32-bit in labs)

基本指令集	描述
RV32I	32-bit integer instruction set
RV32E	Subset of RV32I, for embedded scenarios
RV64I	64-bit integer instruction set，compatible with RV32I
RV128I	64-bit integer instruction set，compatible with RV32I and RV64I

扩展指令集	描述
M	Integer Multiply/Divide extension
A	Atomic extension
F	Single-precision floating-point extension
D	Double-precision floating-point extension
C	Compressed extension
…	Others

RARS - Quick Start(1)

- The Code area is reserved for your RISC-V assembly program editing.
- The Register area displays latest register content
  - Arithmetic operands are in registers
  - 32, 32-bit integer registers in RISC-V (x0~x31).
  - In RARS, registers are displaced using alternative ABI name (more meaningful)



RARS - Quick Start(2)

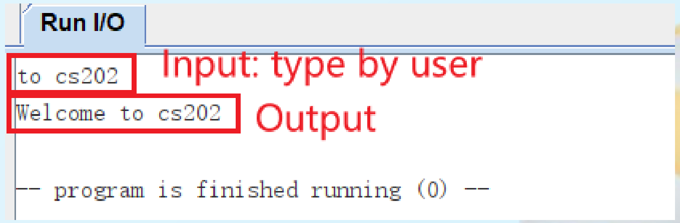
- **Data declaration**
  - Data declaration section starts with “**. data**”.
  - The declaration means **a piece of memory is required to be allocated**. The declaration usually includes **lable** (name of address on this meomory unit), **size**(optional), and **initial value**(optional).
- **Code definition**
  - Code definition starts with “**.text**”, includes **basic instructions, extended instructions, labels of the code**(optional). At the end of the code, “**exit**” system service should be called.
- **Comments**
  - Comments start from “#” till the end of current line
- **Running in Rars**
  - Edit assembly codes
  - Assemble the current file
  - Run the current program

✓Run step by step

li: a pseudo code that load immediate value into register
la: a pseudo code that register to label’s address

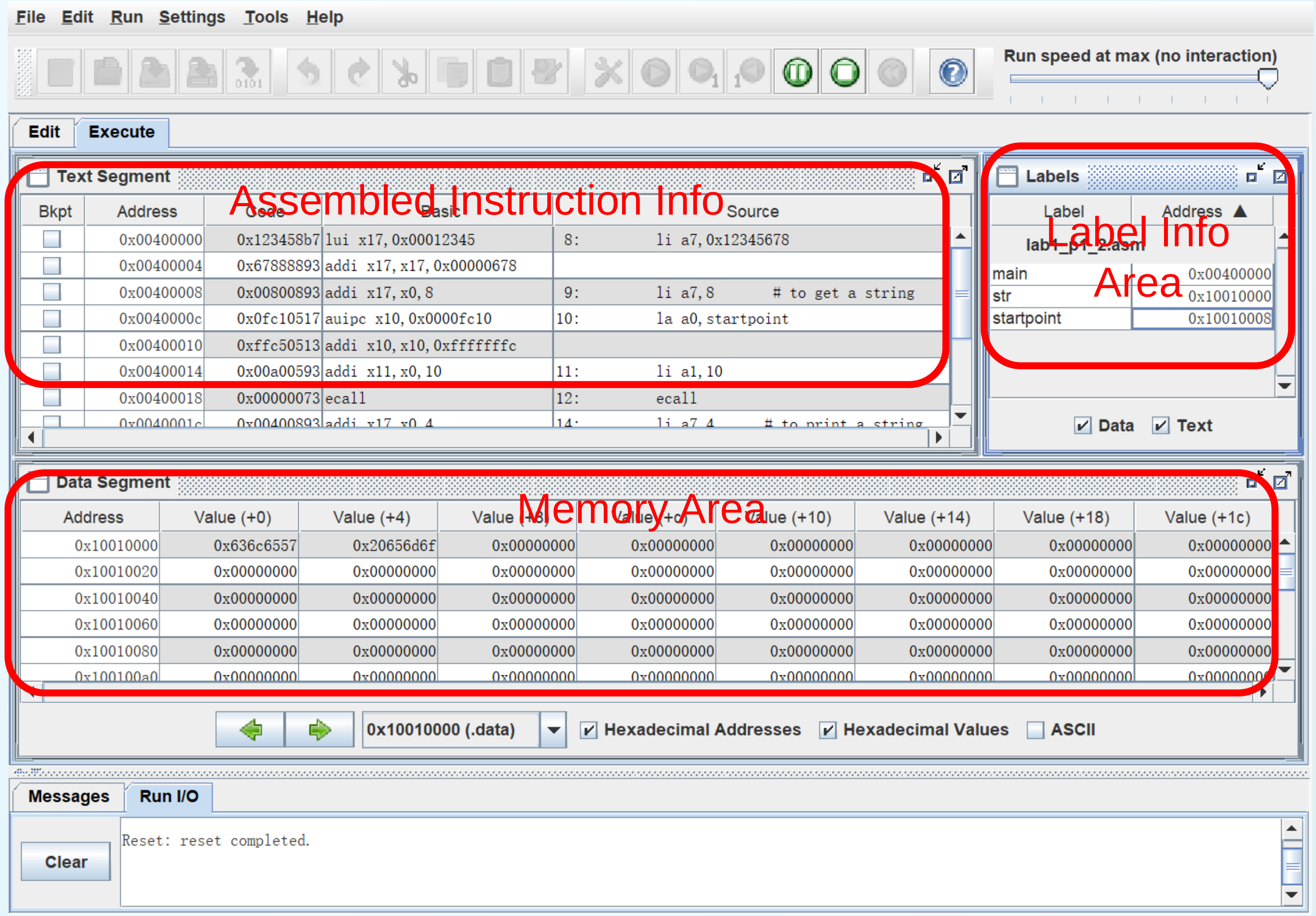
```
# Piece 2-2
.data
        str:            .ascii    "Welcome "
        startpoint:  .space  9

.text
main:
        li a7,8         # to get a string
        la a0,startpoint
        li a1,10
        ecall

        li a7,4         # to print a string
        la a0,str
        ecall

        li a7,10        # to exit
        ecall
```

Show output on bottom of the console:



RARS - Quick Start(3)

- some instruction source program are translated to multiple RISC-V basic instructions (in the Basic column)
  - because they are pseudo instruction (basically syntactic sugar), converted to basic instruction and 32-bit machine codes by the assembler.
- Label info area can be activate by Settings->Show Labels Window
- The data declared in .data section can be found in Memory Area
  - See more details later

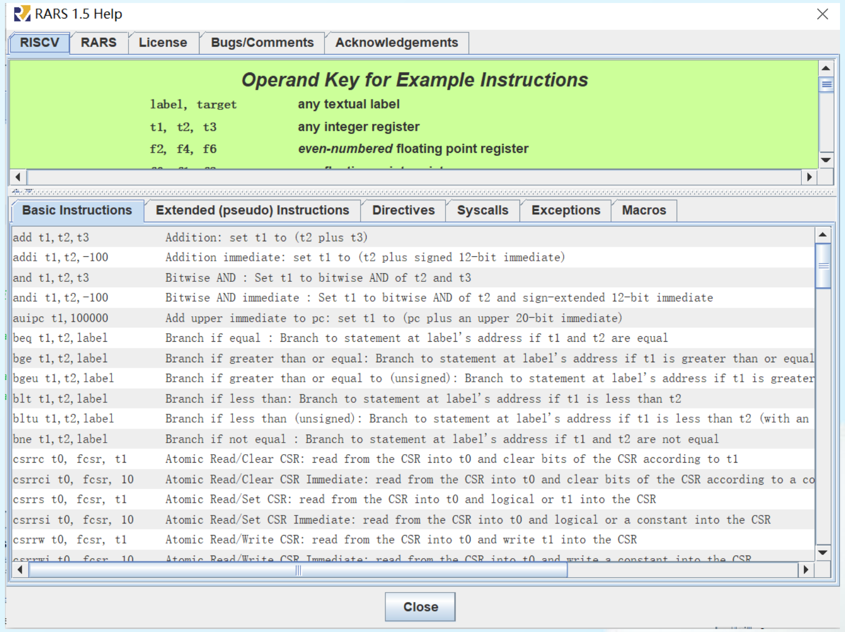


RARS - Instructions

Some RISC-V assembly language instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add x5, x6, x7	x5 = x6 + x7	Three register operands; add
	subtract	add x5, x6, x7	x5 = x6 - x7	Three register operands; subtract
Data transfer	load word	lw x5, 40(x6)	x5 = Mem[x6 + 40]	Word from memory to register
	store word	sw x5, 40(x6)	Mem[x6 + 40] = x5	Word from register to memory
Logical	and	and x5, x6, x7	x5 = x6 & x7	Three register operands; bit-by-bit AND
Shift	shift left logical	sll x5, x6, x7	x5 = x6 << x7	Shift left by register
Conditional branch	branch if equal	beq x5, x6, 100	if(x5 == x6) go to PC+100	PC-relative branch if registers equal
Unconditional branch	jump and link	jal x1, 100	x1 = PC + 4; go to PC+100	PC-relative procedure call

More instructions can be found in RARS Help



RARS - System Call

- A number of system services, mainly for input and output, are available in Rars.
- Example: display a string on the console(a7 = 4) and exit the program(a7 = 10).

```
#RISC-V piece 2-1
.data
str: .asciz "Hello,RISC-V"

.text
li a7,4
la a0,str
ecall

li a7,10
ecall
```

Table of Available Services

Name	Number	Description	Inputs	Ouputs
PrintInt	1	Prints an integer	a0 = integer to print	N/A
PrintFloat	2	Prints a floating point number	fa0 = float to print	N/A
PrintDouble	3	Prints a double precision floating point number	fa0 = double to print	N/A
PrintString	4	Prints a null-terminated string to the console	a0 = the address of the string	N/A
ReadInt	5	Reads an int from input console	N/A	a0 = the int
ReadFloat	6	Reads a float from input console	N/A	fa0 = the float
ReadDouble	7	Reads a double from input console	N/A	fa0 = the double
ReadString	8	Reads a string from the console	a0 = address of input buffer a1 = maximum number of characters to read	N/A
Sbrk	9	Allocate heap memory	a0 = amount of memory in bytes	a0 = address to the allocated block
Exit	10	Exits the program with code 0	N/A	N/A
PrintChar	11	Prints an ascii character	a0 = character to print (only lowest byte is considered)	N/A
ReadChar	12	Reads a character from input console	N/A	a0 = the character

Show output on bottom of the console:

Run I/O

```
Hello,RISC-V
-- program is finished running (0) --
```

Tip: display all the system services information in “Help” of Rars.

Data transfer: load & store

- In RISC-V, memory could **ONLY** be accessed by data transfer instructions.
- In RISC-V, data must be in registers to perform arithmetic.
- Unit Conversion
  - ✓ 1 word = 32bit = 2*half word(2*16bit) = 4* byte(4*8bit)

Name	Example	Comments
32 registers	x0 - x31	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0.
2^{30} memory words	Memory[0], Memory[4], ...,	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential word accesses differ by 4. Memory holds data structures, arrays, and spilled registers.

Data transfer: load(1)

- Load: transfer data to register

Some RISC-V load instructions(including pseudo code)

Mnemonic	Instruction	Example	Meaning	Comments
lw	Load word	lw x5, 40(x6)	x5 = Mem [x6 + 40]	Word from memory to register
lb	Load byte	lb x5, 40(x6)	x5 = Mem [x6 + 40]	Byte from memory to register
lui	Load upper immediate	lui x5, 0x12345	x5 = 0x12345000	Load 20-bit constant shifted left 12 bits
la	Load address	la x5, label1	x5 = label1's address	Set x5 to label1's address

- lui (Load upper immediate) loads 20-bit immediate constant shifted left 12 bits, can be used to load large constant

lui x17, 0x00012345	8: li a7, 0x12345678
addi x17, x17, 0x00000678	

- la (load address) is a extended (pseudo) instruction, which is implemented by two basic instructions: auipc (add upper immediate to PC), addi (add immediate).

- auipc (U-type): to add 20-bit upper immediate to PC; to write sum to register. a's address = 1001 0000

0x0040000c	0x0fc10397	auipc x7, 0x0000fc10	10: la t2, a
0x00400010	0xff438393	addi x7, x7, 0xfffffff4	

Data transfer: load(2)

- In addition to word data load (lw), RISC-V has byte, halfword data transfers (lb, lh), data is copied to the **low byte position** of register.
- Data is **sign-extended** to register.

```
# Piece 2-4
.data
        a:     .word 0x12345678
        b:     .word 0x9abcdef0

.text
main:
        lw t0, a      # load word
        lw t1, b

        lb t0, a      # load byte
        lb t1, b

        lh t0, a      #load halfword
        lh t1, b
```

t0	5	0x00000000
t1	6	0x00000000

t0	5	0x12345678
t1	6	0x9abcdef0

t0	5	0x00000078
t1	6	0xfffffff0

t0	5	0x00005678
t1	6	0xffffdef0

- a: 12345678_{hex} = $0001\ 0010\ 0011\ 0100\ 0101\ 0110\ 0111\ 1000_{two}$
- b: $9abcdef0_{hex}$ = $1001\ 1010\ 1011\ 1100\ 1101\ 1110\ 1111\ 0000_{two}$

- When executing **lb** instruction

78_{hex} = $\underline{0}111\ 1000_{two}$

sign-extended

0000 0000 0000 0000 0000 0000 $0111\ 1000_{two}$ = 00000078_{hex}

$f0_{hex}$ = $\underline{1}111\ 0000_{two}$

sign-extended

1111 1111 1111 1111 1111 1111 $1111\ 0000_{two}$ = $fffffff0_{hex}$

Data transfer: store(1)

- Store: transfer data from register to memory

Some RISC-V store instructions

Mnemonic	Instruction	Example	Meaning	Comments
sw	Store word	sw x5, 40(x6)	Mem [x6 + 40] = x5	Word from register to memory
sb	Store byte	sb x5, 40(x6)	Mem [x6 + 40] = x5	Byte from register to memory

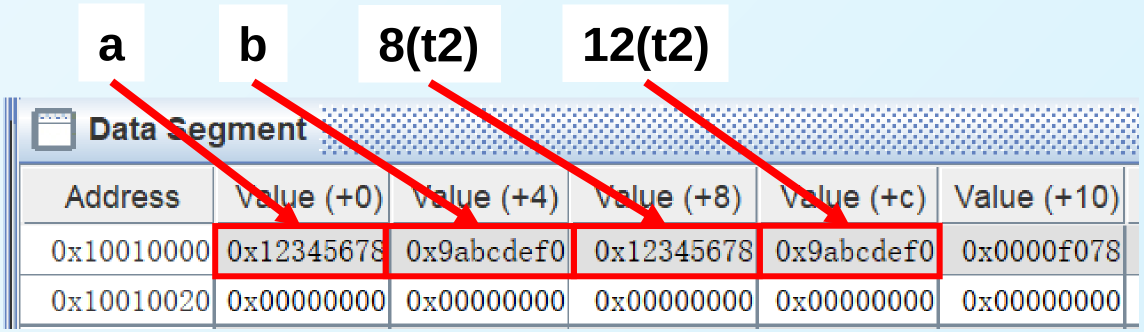
- **Question:** Is it necessary to implement “sa” instruction (store address)?

Why? If it is necessary to implement “sa”, how to do it?

Data transfer: store(2)

```
# Piece 2-5
.data
        a:      .word 0x12345678
        b:      .word 0x9abcdef0

.text
main:
        lw t0, a       # load word
        lw t1, b
        la t2, a

        sw t0, 8(t2)           # store word
        sw t1, 12(t2)

        li a7,10            # to exit
        ecall
```

- t2: 10010000_{hex}
- 8(t2): $10010000_{hex} + 8_{ten} = 10010008_{hex}$
- 12(t2) : $10010000_{hex} + 12_{ten} = 1001000c_{hex}$
- A word occupies 4 bytes, so t2+8 and t2+12 are multiples of 4.

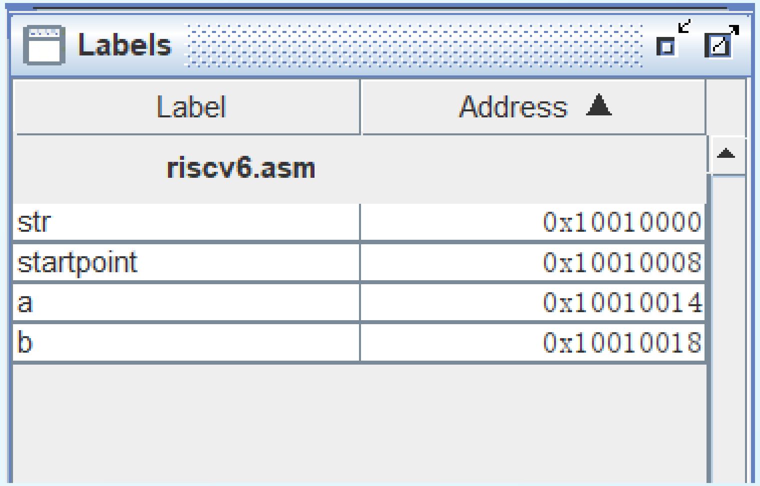


Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x10010000	0x12345678	0x9abcdef0	0x12345678	0x9abcdef0	0x0000f078
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Address alignment (1)

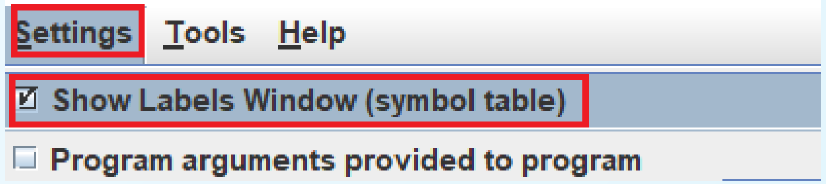
➤ The value of “**label**” is determined by the Assembler according to the assembly source code.

```
# Piece 2-6
.data
        str:            .ascii    "Welcome "
        startpoint: .space  9
        a:              .word 0x12345678
        b:              .word 0x9abcdef0
.text
main:
        # ......

        li a7,10            # to exit
        ecall
```



Question: Why a’s address is 0x10010014 but not 0x10010011?

➤ Tip: to show labels window, go to “Settings” menu.



Address alignment (2)

- The address need to be **calculated by Baseline + offset** (Using the sum of the baseline address and offset as memory address).
  - ✓ Load the word from the memory unit whose address is the sum of 4 and the value in register t0 to register t2. lw t2, 4(t0)
  - ✓ Store half-word in register t2 to memory unit whose address is the sum of -12 and the value in register t0. sh t2, -12(t0)
- if it's lw, a word occupies 4 bytes, so t2+8 is multiples of 4.
- if it's sh, a half-word occupies 2 bytes, so t2-12 is multiples of 2.

- **Question**: Run the piece of codes on right hand, observe the executed results and explain the reason.

```
# Piece 2-7
.data
        a:      .word 0x12345678
        b:      .word 0x9abcdef0

.text
main:
        lw t0, a        # load word
        lw t1, b
        la t2, a

        sb t0, 8(t2)            # store byte
        sb t1, 9(t2)

        sw t0, 10(t2)           # store word
        sw t1, 14(t2)

        sh t0, 18(t2)           # store halfword
        sh t1, 20(t2)

        li a7,10                # to exit
        ecall
```

Data interpretation(1)

```
name:        storage_type    value(s)
```

```
example

var1:      .word   3       # create a single integer:
                           #variable with initial value 3

array1:    .byte   'a','b' # create a 2-element character
                           # array with elements initialized:
                           # to  a  and  b

array2:    .space  40      # allocate 40 consecutive bytes,
                           # with storage uninitialized
                           # could be used as a 40-element
                           # character array, or a
                           # 10-element integer array;
                           # a comment should indicate it.
```

```
.data
        var1:       .word  3
        array1:   .byte 'a', 'b'
```

Labels

Label	Address ▲
riscv6.asm	
var1	0x10010000
array1	0x10010004

Data Segment

Address	Value (+0)	Value (+4)
0x10010000	0x00000003	0x00006261

Display in ASCII code

Data Segment

Address	Value (+0)	Value (+4)
0x10010000	\0 \0 \0 .	\0 \0 b a

Data interpretation(2)

macro_print_str.asm program can be found in Last page

- while calculate the data, if the instruction ends with “**u**” means the data are treated as **unsigned** integer, else the data are treated as **signed by default**.
- slt t1,t2,t3

  set less than: if t2 is less than t3, then set t1 to 1 else set t1 to 0.
- sltu t1,t2,t3

  set less than unsigned: if t2 is less than t3 using unsigned comparison, set t1 to 1 else set t1 to 0.

```
# Piece 2-8
.include "macro_print_str.asm"
.data
.text
main:
	print_string("\n -1 is less than 1 using slt (1 for yes and 0 for no): ")
	li t0,-1
	li t1,1
	slt a0,t0,t1
	li a7,1
	ecall

	print_string("\n -1 is less than 1 using sltu (1 for yes and 0 for no): ")
	sltu a0,t0,t1
	li a7,1
	ecall

	end
```

Run I/O

```
-1 is less than 1 using slt (1 for yes and 0 for no): 1
-1 is less than 1 using sltu (1 for yes and 0 for no): 0
-- program is finished running (0) --
```

- RISC-V also has “unsigned byte” loads (lbu) which zero extends to fill register, and also has lwu and lhu.

Data interpretation(3)

- Run the piece of codes on right hand, answer the questions.
  - Q1. What's the data stored in register a0 after execute "lw a0, tdata"?
  - Q2. What are the two display result?
  - Q3. Is the 2nd "lw a0, tdata" instruction after print_string("\n") redundant? If delete, what will be displayed, why?
- Tip: system call
  - code **1**: display data in **a0** as **signed** decimal value
  - code **36**: display data in **a0** as **unsigned** decimal value

```
# Piece 2-9
.include "macro_print_str.asm"
.data
	tdata: .word 0xFFFFFFFF
.text
main:
	lw a0, tdata
	li a7, 1
	ecall

	print_string("\n")
	lw a0, tdata
	li a7, 36
	ecall

	li a7, 10
	ecall
```

Data interpretation(4)

- Run the two pieces of codes on right hand, answer the questions.
  - Q1: What are the values stored in the register a0 after the operation of ‘lb’ and ‘lbu’?
  - Q2: using “-1” as initial value of tdata instead of “0x80”, answer Q1 again.

```
# Piece 2-10
.include "macro_print_str.asm"
.data
	tdata: .byte 0x80
.text
main:
	lb a0, tdata
	li a7, 1
	ecall

	print_string("\n")
	lb a0, tdata
	li a7, 36
	ecall

	end
```

```
# Piece 2-11
.include "macro_print_str.asm"
.data
	tdata: .byte 0x80
.text
main:
	lbu a0, tdata
	li a7, 1
	ecall

	print_string("\n")
	lbu a0, tdata
	li a7, 36
	ecall

	end
```

Practice 1

- Use RISC-V assembly language to program and realize the following functions on Rars: Using system calls to get the sid which has 8 numbers from input, print out the string: Welcome XXXXXXXX to RISC-V World (XXXXXXXX is an 8-digit number)
  - ✓ 1-1. complete the codes on the right hand, move the string " to RISC-V World" from the memory unit addressed by "e1" to the memory unit addressed by the sum of 8 and "sid".
  - ✓ 1-2. Is there any other way to implement the function?
  - ✓ 1-3. Which method would get better performance: 1-1 or 1-2?
- Tip 1: While get and put string by syscall, the end of string is **"\0"** which means getting a string would add a "\0" at the end of string, print a string would end with "\0"
- Tip 2: The difference between "**ascii**" and "**asciz**" is that "asciz" would add "\0" at the end of the string while "ascii" would not.

```
# Piece 2-12
.data
        str:        .ascii   "\nWelcome "
        sid:        .space  9
        e1:         .asciz   " to RISC-V
World"
.text
main:
        li a7, 8        # to get a string
        la a0, sid
        li a1, 9
        ecall

        #complete code here

        li a7, 4        # to print a string
        la a0, str
        ecall

        li a7, 10       # to exit
        ecall
```

Practice 2

- Run the code on the right hand, answer the questions.
  - ✓ 2-1. What's the value of label alice?
  - ✓ 2-2. What's the value of label tony?
  - ✓ 2-3. What's the output after execute the system call on line 22?

lab1-practice4.asm

```
1  .data
2          name:   .space 16       # malloc 16 bytes, not initialize
3          mick:   .ascii "Mick\n" # malloc 4+1 = 5 bytes
4          alice:  .asciz "Alice\n"       ##### What's the value of alice?
5          tony:   .asciz "Tony\n" ##### What's the value of tony?
6          chen:   .asciz "Chen\n"
7
8  .text
9  main:
10         la t0, name
11         la t1, mick
12         sw t1, (t0)             # get the value of t0; use it as the address of a piece of memory
13         la t1, alice
14         sw t1, 4(t0)            # baseline: the content of t0; offset: 4
15         la t1, tony
16         sw t1, 8(t0)
17         la t1, chen
18         sw t1, 12(t0)
19
20         li a7, 4
21         lw a0, 0(t0)
22         ecall                   # What's the output while this system call is done?
23
24         li a7, 10
25         ecall
```

Tip 1 ： macro_print_str.asm

- Get help of definition and usage about macro from help page.
- While using the macro, put this file to the same directory as the file which use the macro.
- Name this file as "macro_print_str.asm"

```
.macro print_string(%str)
	.data
		pstr:	.asciz	%str
	.text
		la a0,pstr
		li a7,4
		ecall
.end_macro

.macro end
	li a7,10
	ecall
.end_macro
```