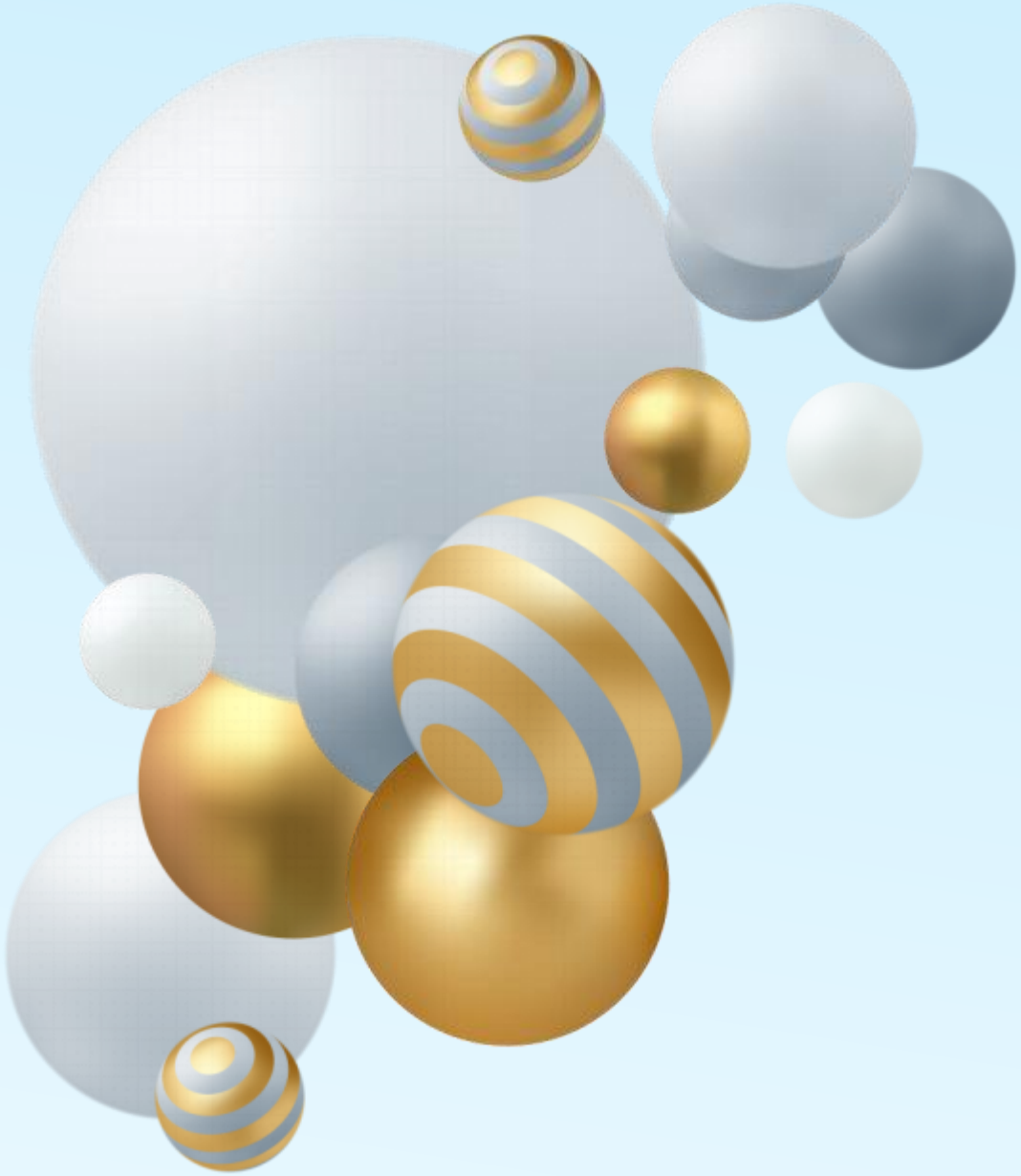
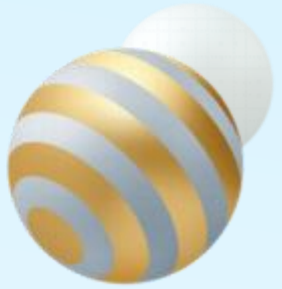


Computer Organization

Lab5 RISC-V instructions(3)

Instruction Format & Directives



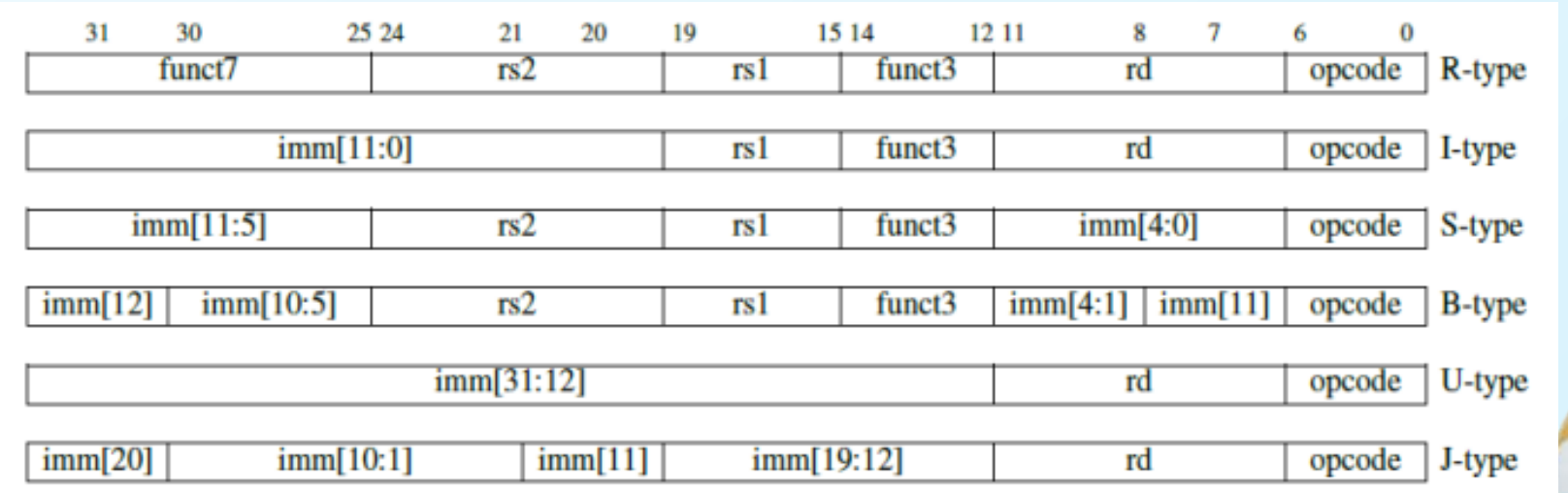
Topics

- RISC-V Instruction format
  - ✓ Basic type
  - ✓ Immediate data
- Assembler - Directives
  - ✓ .macro & .end_macro
  - ✓ .align
  - ✓ .globl (.global) vs .extern
- Practice

RISC-V instruction format: Basic type(1)

- 6 basic instruction format types: R, I, S, B, U, J
  - ✓ R-type: for operation between registers
  - ✓ I-type: used by arithmetic operands with one constant operand, and by load instructions
  - ✓ S-type: for storing operation
  - ✓ B-type: for conditional branch
  - ✓ U-type: for long immediate
  - ✓ J-type: for unconditional branch

Basic instruction format in RISC-V



RISC-V instruction format: Basic type(2)

- I-type
  - addi: add immediate

addi t1, t0, 1

imm[11:0]	rs1	000	rd	0010011
1_{ten} = 000000000001	t0(x5): 00101		t1(x6): 00110	

  - Machine code: $00000000000100101000001100010011_{two} = 00128313_{hex}$

- S-type
  - sw: store word

sw t1, 0(t2)

imm[11:5]	rs2	rs1	010	imm[4:0]	0100011
0_{ten} = 0000000	t1(x6): 00110	t2(x7): 00111		00000	

  - Machine code: $00000000011000111010000000100011_{two} = 0063a023_{hex}$

RISC-V instruction format: Basic type(3)

- la (load address) is implemented by two basic instructions: **auipc** and addi
- auipc (U-type): to add **20-bit** upper immediate to PC; to write sum to register.

auipc t2, 0x0000fc10

imm[31:12]	rd	opcode
0x000**0fc10**: **0000_1111_1101_0001_0000**	t2(x7): 00111	0010111

- ✓ Machine code: $00001111110100010000001110010111_{two} = 0fc10397_{hex}$
- ✓ Immediate data: 0x0fc10000
- ✓ t2(x7) = PC (0x0040000c) + immediate data (0x0fc10000) = 0x1001000c

- addi x7, x7, 0xffffffff4
  - ✓ t2(x7) = t2 + 0xffffffff4 = 0x1001000c + 0xffffffff4 = 0x10010000

Labels

Label	Address
lab5-piece5-0.asm	
main	0x00400000
a	0x10010000
b	0x10010004

Address	Code	Basic	Source
0x00400000	0x0fc10297	auipc x5, 0x0000fc10	8: lw t0, b
0x00400004	0x0042a283	lw x5, 4 (x5)	
0x00400008	0x00128313	addi x6, x5, 1	9: addi t1, t0, 1
0x0040000c	0x0fc10397	auipc x7, 0x0000fc10	10: la t2, a
0x00400010	0xff438393	addi x7, x7, 0xffffffff4	
0x00400014	0x0063a023	sw x6, 0 (x7)	11: sw t1, 0(t2)
0x00400018	0x00700333	add x6, x0, x7	13: mv t1, t2

```
# Piece 5-0
.data
        a:      .word 0x1111
        b:      .word 0x5555
.text
main:
        lw t0, b
        addi t1, t0, 1
        la t2, a
        sw t1, 0(t2)
        mv t1, t2
```

RISC-V instruction format: Immediate data(1)

- If we want to calculate a = b + 1, we can use **addi** instruction. addi t0, t1, 1
- addi is of I-type

- For I-type instructions, imm[11:0] can hold values in range **[$-2048_{(10)}$, $+2047_{(10)}$]**.

- If we want to calculate a = b + **2049**, or greater numbers, what should we do?
- We can use instructions in U-type format. lui: Load Upper Immediate

- For U-type instructions, immediate data occupies 20 bits, we use them as upper 20 bits of a long immediate data. And what about the other 12 bits (32-20 = 12) ? We can use another addi instruction to add the low 12 bits.

```
# Piece 5-1
lui a0, 0x12345          # a0 = 0x12345000
addi a0, a0, 0x678       # a0 = 0x12345678

li a7, 34
ecall
```

RISC-V instruction format: Immediate data(2)

- Run the demo #Piece 5-2, will the output be 0x12345abc?
- Run the demo #Piece 5-3, answer the questions?
  - ✓ Q1: What's the output? Are the printed numbers the same with your expectation?
  - ✓ Q2: While dvalue2 is bigger than dvalue1, why the 2nd number is not bigger than the 1st number?

```
# Piece 5-2
.text
main:
        lui a0, 0x12345     # a0 = 0x12345000
        addi a0, a0, 0xabc      # a0 = 0x12345abc ?
        li a7, 34
        ecall
```

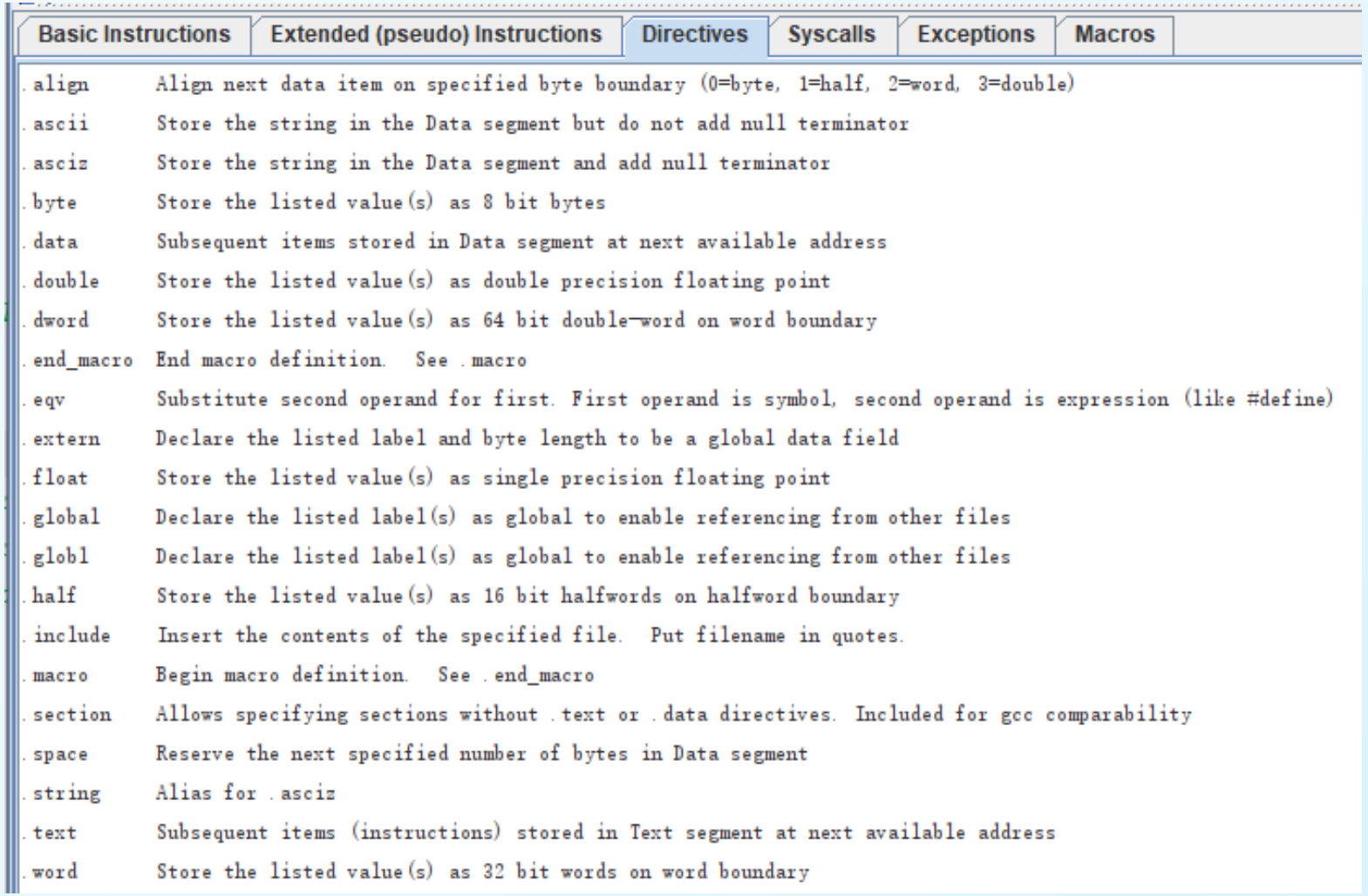
```
# Piece 5-3
.include "macro_print_str.asm"
.data
        dvalue1: .word 0x00000abc
        dvalue2: .word 0x7fffffff
        .text
main:
        lui a0, 0x12345
        lw t1, dvalue1
        add t0, a0, t1
        # 1st number
        mv a0, t0
        li a7, 1
        ecall
        print_string("\n")

        lui a0, 0x12345
        lw t1, dvalue2
        add y0, a0, t1
        # 2nd number
        mv a0, t0
        li a7, 1
        ecall
        end
```

RISC-V instruction format: Overflow

- In RISC-V, arithmetic overflow are checked by software, that is to say, you should use your codes to check whether an overflow occurs.
- Run the demo on right hand, change the values of dvalue1 and dvalue2, and check for overflow occurrence to each group of values.
  - ✓ Group 1. dvalue1: 0x7fffffff; dvalue2: 0x00000001
  - ✓ Group 2. dvalue1: 0x7fffffff; dvalue2: 0xffffffff
  - ✓ Group 3. dvalue1: 0x7fffffff; dvalue2: -1
  - ✓ Group 4. dvalue1: 0x7fffffff; dvalue2: 0x80000000
  - ✓ Group 5. dvalue1: 0x7fffffff; dvalue2: 0x7fffffff
  - ✓ Group 6. dvalue1: 0x80000001; dvalue2: 0x80000001
  - ✓ Group 7. dvalue1: 0x80000001; dvalue2: 1

```
# Piece 5-4
.include "macro_print_str.asm"
.data
        dvalue1: .word 0x02
        dvalue2: .word 0x0f
.text
        lw t1, dvalue1
        lw t2, dvalue2
        add t0, t1, t2              # add two values
        slti t3, t2, 0              # t3 = (t2 < 0)
        slt t4, t0, t1              # t4 = (t0 < t1), thst is, (t1 + t2 < t1)

        mv a0, t0           # print the sum
        li a7, 1
        ecall
        bne t3, t4, overflow    # overflow if (t2 < 0) && (t1 + t2 >= t1)
                                # or if (t2 >= 0) && (t1 + t2 < t1)
        print_string("\nNo overflow occured.")
        jal exit
overflow:
        print_string("\nOne overflow occured.")
exit:
        end
```

Directives of Rars



Basic Instructions | Extended (pseudo) Instructions | Directives | Syscalls | Exceptions | Macros

.align	Align next data item on specified byte boundary (0=byte, 1=half, 2=word, 3=double)
.ascii	Store the string in the Data segment but do not add null terminator
.asciz	Store the string in the Data segment and add null terminator
.byte	Store the listed value(s) as 8 bit bytes
.data	Subsequent items stored in Data segment at next available address
.double	Store the listed value(s) as double precision floating point
.dword	Store the listed value(s) as 64 bit double-word on word boundary
.end_macro	End macro definition. See .macro
.eqv	Substitute second operand for first. First operand is symbol, second operand is expression (like #define)
.extern	Declare the listed label and byte length to be a global data field
.float	Store the listed value(s) as single precision floating point
.global	Declare the listed label(s) as global to enable referencing from other files
.globl	Declare the listed label(s) as global to enable referencing from other files
.half	Store the listed value(s) as 16 bit halfwords on halfword boundary
.include	Insert the contents of the specified file. Put filename in quotes.
.macro	Begin macro definition. See .end_macro
.section	Allows specifying sections without .text or .data directives. Included for gcc comparability
.space	Reserve the next specified number of bytes in Data segment
.string	Alias for .asciz
.text	Subsequent items (instructions) stored in Text segment at next available address
.word	Store the listed value(s) as 32 bit words on word boundary

Directives: .macro & .end_macro (1)

- **.macro**
  - A **pattern-matching** and **replacement** facility that provide a simple mechanism to name a frequently used sequence of instructions.
  - **Programmer invokes** the macro.
  - **Assembler replaces** the macro call with the corresponding sequence of instructions.

- **Macros vs** procedures
  - **Same:** permit a programmer to create and name a new abstraction for a common operation.
  - **Difference:** Unlike procedures, macros do not cause a subroutine call and return when the program runs since a macro call is replaced by the macro's body when the program is assembled.

Directives: .macro & .end_macro (2)

- Assembler replaces the macro call with the corresponding sequence of instructions.
  - ✓ Q1: What's the difference between macro and procedure?
  - ✓ Q2: While save the procedure's definition (#piece 5-5 on the right hand) in a ***.asm file, and assemble it, what's the assembly result? Is the procedure definition file executable?
  - ✓ Q3: While save the macro's definition (#piece 5-6 on the right hand) in a ***.asm file, and assemble it, what's the assembly result? Is the macro definition file executable?

```
# Piece 5-5
.text
print_string:
        addi sp, sp, -8
        sw ra, 4(sp)
        sw a0, (sp)

        li a7, 4
        ecall

        lw a0, (sp)
        lw ra, 4(sp)
        addi sp, sp, 8

        jr ra
```

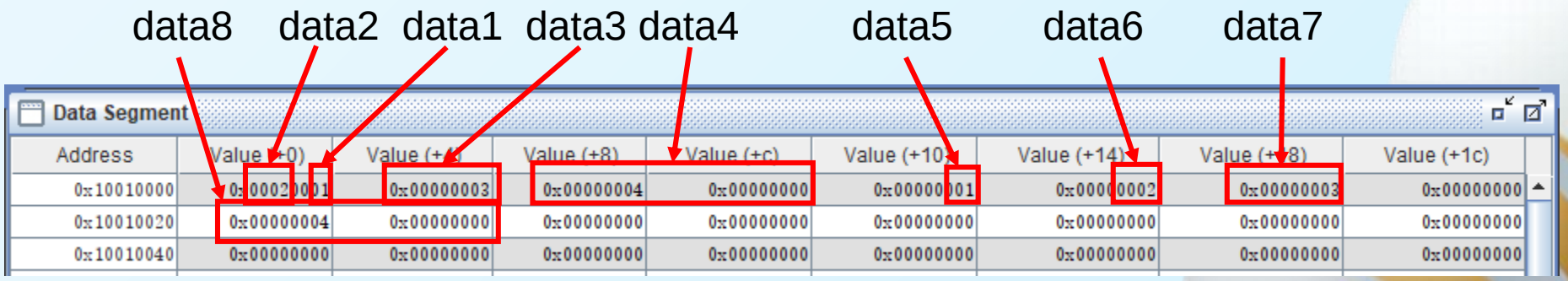
```
# Piece 5-6
.macro print_string(%str)
.data
        pstr:   .asciz   %str
.text
        addi sp, sp, -8
        sw ra, 4(sp)
        sw a0, (sp)

        la a0, pstr
        li v7, 4
        ecall

        lw a0, (sp)
        lw ra, 4(sp)
        addi sp, sp, 8

.end_macro
```

Directives: .align (1)

- **.align**
  - Align next data item on specified byte boundary.
  - 0=byte, 1=half, 2=word, 3=double
  - Run the demo on right hand, observe the address of each label, and answer the questions.
  - Q1. Why the address of data2 is 0x10010002, but not 0x10010001?
  - Q2. How many space(bytes) does data4 occupy?
  - Q3. Why the address of data6 is 0x10010014, but not 0x10010012?

```
# Piece 5-7
.data
        data1: .byte 1
        data2: .half 2
        data3: .word 3
        data4: .dword 4
.align 2
        data5: .byte 1
.align 2
        data6: .half 2
.align 3
        data7: .word 3
.align 3
        data8: .dword 4
```

Label	Address ▲
lab5-piece5-7.asm	
data1	0x10010000
data2	0x10010002
data3	0x10010004
data4	0x10010008
data5	0x10010010
data6	0x10010014
data7	0x10010018
data8	0x10010020



Directives: .align (2)

- Run the two demos on right hand, and answer the questions.
  - ✓ Q1. Which demo(s) would invoke an exception "*** address not aligned to word boundary 0x10010007"?
  - ✓ Q2. Which instruction would invoke the exception? lb, sw, lw, or sb?
  - ✓ Tips: While transfering data, the address of data in memory is required to be aligned according to the bit width of data.
  - ✓ Q3. If adding ".align 2"in this demo, can this kind of error be avoided? And where should we place this directive? Position A, B, or C?

```
# Piece 5-8
.data
        # Position A
        str1:  .ascii  "Welcome“
        # Position B
        str2:  .ascii  "to“
        # Position C
        str3:  .asciz  "RISC-V World"
.text
        la t0, str2
        lb t1, (t0)
# change lowercase letter to uppercase
        addi t1, t1, -32
        sw t1, (t0)

        la a0, str1
        li a7, 4
        ecall

        li a7, 10
        ecall
```

```
# Piece 5-9
.data
        # Position A
        str1:  .ascii  "Welcome“
        # Position B
        str2:  .ascii  "to“
        # Position C
        str3:  .asciz  "RISC-V World"
.text
        la t0, str2
        lw t1, (t0)
        addi t1, t1, -32
        sb t1, (t0)

        la a0, str1
        li a7, 4
        ecall

        li a7, 10
        ecall
```

Directives: .global(.globl) & .extern (1)

- **.include**
  - ✓ Insert the contents of the specified **file**, put filename in quotes
- **.globl**
  - ✓ Declare the listed **label**(s) as global to enable referencing from other files
- **.extern**
  - ✓ Declare the listed **label** and byte length to be a global **data** field
- **Local label**
  - ✓ A label referring to an object that can be used **ONLY within the FILE** in which it is defined.
- **External label**
  - ✓ A label referring to an object that can be referenced from **FILE other than the one in which it is defined**.

Directives: .global & .extern demo(1)

- Q1. Assemble “read_caller.asm” and run the codes, what is the result?
- Q2. What is the address of label read_str? What segment does it locate?
- Q3. If we change the statement “**.extern read_str 20**” to “**read_str: .space 20**”, what is the running result? And what is the address of label read_str? What segment does it locate?

```
##   “read_caller.asm" ##
.include "read_callee.asm"
.data
    str_caller:       .asciz "\nIt's in read_caller.\n"
.text
.globl main
main:
        li a7, 8
        la a0, read_str
        li a1, 20
        ecall

        jal read_callee

        li a7,10
        ecall
```

```
##   “read_callee.asm" ##
.data
        .extern read_str   20
         #read_str:.space 20
        str_callee:  .asciz  "\nIt's in read_callee.\n"
.text
read_callee:
        li a7, 4
        la a0, str_callee
        ecall
        la a0, read_str
        ecall

        jr ra
```

Directives: .global & .extern demo(2)

- Q1. How many “default_str” are defined in “print_callee.asm”? What is(are) the address(es)?
- Q2. Is the running result the same as the screenshot in the bottom right corner?
- Q3. While executing the instruction “**la a0, default_str**” in these two files, which “default_str” is used in each file?
- Q4. What will happen if an external variable has the same name with a local variable?

```
## "print_caller.asm" ##
.include "print_callee.asm"
.data
    str_caller:       .asciz "It's in print_caller.\n"
.text
.globl main
main:
        jal print_callee

        li a7, 4
        la  a0, str_caller
        ecall
        la a0, default_str   ### Which one?
        ecall

        li a7,10
        ecall
```

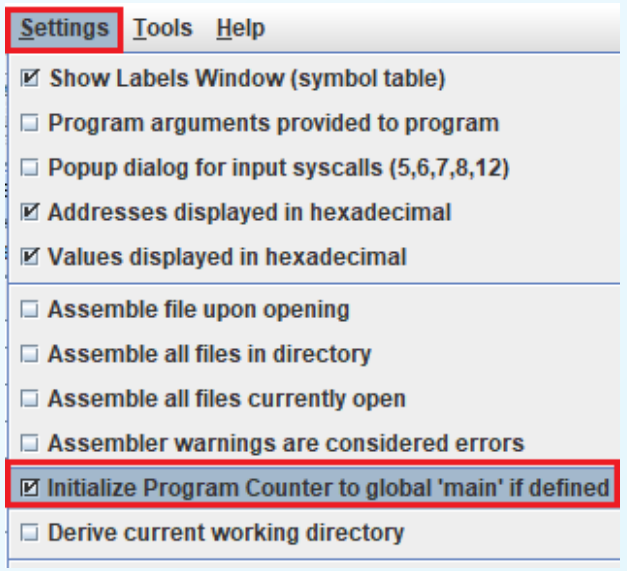
```
## "print_callee.asm" ##
.data
        .extern default_str        20
        default_str: .asciz  "It's the default string in data seg\n"
        str_callee:  .asciz  "It's in print_callee.\n"
.text
print_callee:
        li a7, 4
        la a0, str_callee
        ecall
        la a0, default_str    ### Which one?
        ecall

        jr ra
```

```
It's in print_callee.
It's the default string in data seg
It's in print_caller.
It's the default string in data seg
```

Tips on Rars

- To make the instruction labeled by '.global main' as the 1st instruction to run, do the following settings: In Rars menu [**Setting**] -> [Initialize Program Counter to global 'main' if defined].

```
.include "print_callee.asm"
.data
    str_caller: .asciz "It's in print_caller."
.text
.globl main
main:
        jal print_callee
```



Text Segment

Bkpt	Address	Code	Basic		Source
	0x00400010	0x0fc10517	auipc x10, 0x0000fc10	11:	la a0, default_str ### Which one?
	0x00400014	0xff050513	addi x10, x10, 0xfffffff0		
	0x00400018	0x00000073	ecall	12:	ecall
	0x0040001c	0x00008067	jalr x0, x1, 0	14:	jr ra
	0x00400020	0xfe1ff0ef	jal x1, 0xffffffe0	8:	jal print_callee
	0x00400024	0x00400893	addi x17, x0, 4	10:	li a7, 4
	0x00400028	0x0fc10517	auipc x10, 0x0000fc10	11:	la a0, str_caller
	0x0040002c	0x01450513	addi x10, x10, 20		
	0x00400030	0x00000073	ecall	12:	ecall
	0x00400034	0x0fc10517	auipc x10, 0x0000fc10	13:	la a0, default_str ### Which one?
	0x00400038	0xfcc50513	addi x10, x10, 0xffffffcc		
	0x0040003c	0x00000073	ecall	14:	ecall
	0x00400040	0x00a00893	addi x17, x0, 10	16:	li a7, 10
	0x00400044	0x00000073	ecall	17:	ecall

Labels

Label	Address ▲
(global)	
main	0x00400020
default_str	0x10000000

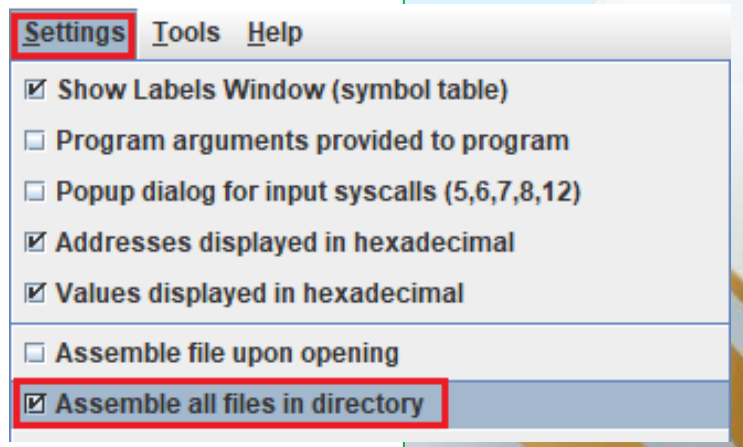
pc		0x00400020

Directives: .global & .extern demo(3)

- Q1. How many “default_str” are defined in “print_callee_e.asm”?
- Q2. While executing “**la a0, default_str**” in these two files, which “default_str” is used in each file?
- Q3. What’s the running result?
- Tips: Store the two files in the same directory, set “Assemble all files in directory”, and then run it.

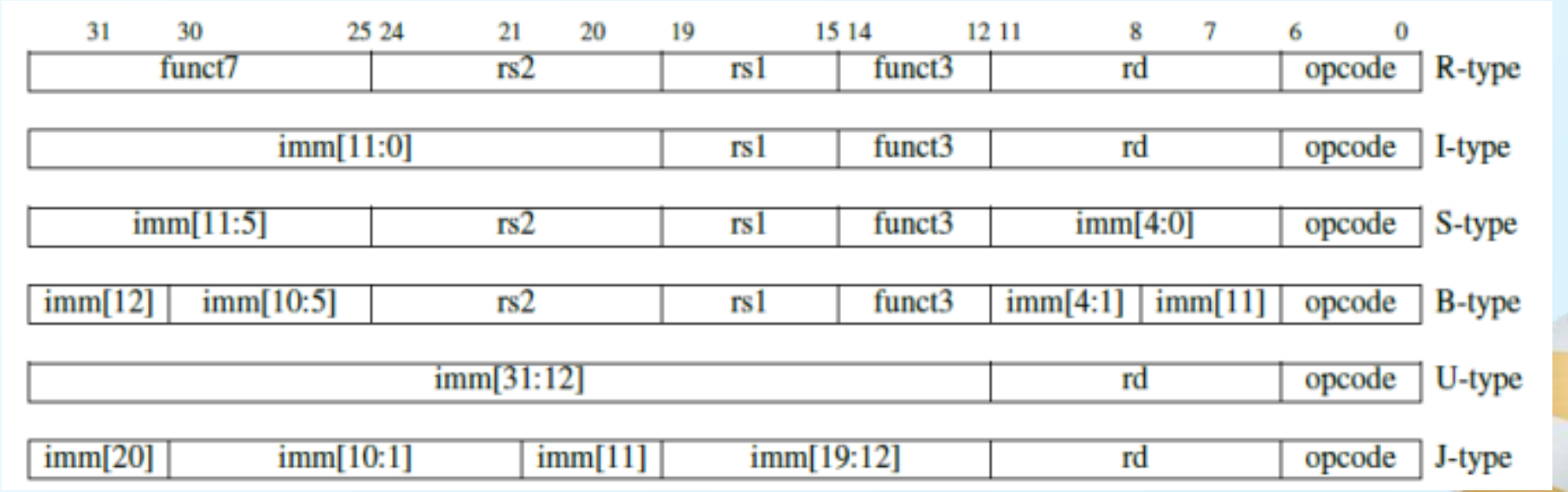
```
## "print_caller_e.asm" ##
.data
    str_caller: .asciz "It's in print_caller.\n"
    data1:      .word 0x64636261   #abcd
.text
.globl  main
main:
        jal print_callee

        la a1, data1
        lw a0, (a1)
        la a1, default_str
        sw a0, (a1)
        li a7, 4
        la a0, str_caller
        ecall
        la a0, default_str
        ecall

        jal print_callee

        li a7, 10
        ecall
```

```
## "print_callee_e.asm" ##
.data
        .extern     default_str   20
        str_callee:         .asciz "It's in print_callee.\n"
        default_str:        .asciz "ABCD\n"
.text
.globl  print_callee
print_callee:
        addi sp, sp, -4
        sw a0, (sp)

        li a7, 4
        la a0, str_callee
        ecall
        la a0, default_str
        ecall

        lw a0, (sp)
        addi sp, sp, 4
        jr ra
```



Practice 1

- Implement in Verilog:
  - ✓ Suppose each instruction is 32 bit wide, and there are 6 types of instruction format, the 6 types of format are R, I, S, B, U, and J, and the specifications to each format are as following.
  - ✓ Suppose opcode for R, I, S, B, U, J are “7’b0000011”, “7’b0000111”, “7’b0001111”, “7’b0001011”, “7’b0011011”, “7’b0011111” respectively.
  - ✓ Please design 6 legal instructions with 6 formats, and extract the immediate data for each type.
  - ✓ Note 1: the immediate data will be sign-extend to a 32-bit register.
  - ✓ Note 2: for R-type, you can handle the immediate data as whatever you want.



Practice 2

- 2-1. Replace the statement “add t0, t1, t2” of **demo piece 5-4** with a “**sub**” instruction, and implement overflow checking function.
- 2-2. Run the demos below, and answer the questions.
  - Q1. Which demo(s) would run without exception?
  - Q2. Which demo(s) would get the output “WelcomeToRISC-VWorld”?

```
# Piece 5-10
.data
        str1:  .ascii  "Welcome"
        str2:  .ascii  "to"
        str3:  .asciz  "RISC-VWorld"
.text
        la t0, str2
        lh t1, (t0)
        addi t1, t1, -32
        sh t1, (t0)

        la a0, str1
        li a7, 4
        ecall

        li a7, 10
        ecall
```

```
# Piece 5-11
.data
        str1:  .ascii  "Welcome"
        .align 2
        str2:  .ascii  "to"
        str3:  .asciz  "RISC-VWorld"
.text
        la t0, str2
        lw t1, (t0)
        addi t1, t1, -32
        sw t1, (t0)

        la a0, str1
        li a7, 4
        ecall

        li a7, 10
        ecall
```

```
# Piece 5-12
.data
        .align 2
        str1:  .ascii  "Welcome"
        str2:  .ascii  "to"
        str3:  .asciz  "RISC-VWorld"
.text
        la t0, str2
        lw t1, (t0)
        addi t1, t1, -32
        sw t1, (t0)

        la a0, str1
        li a7, 4
        ecall

        li a7, 10
        ecall
```

```
# Piece 5-13
.data
        str1:  .ascii  "Welcome"
        str2:  .ascii  "to"
        str3:  .asciz  "RISC-VWorld"
.text
        la t0, str2
        lb t1, (t0)
        addi t1, t1, -32
        sb t1, (t0)

        la a0, str1
        li a7, 4
        ecall

        li a7, 10
        ecall
```