Computer Organization

Lab4 RISC-V instructions(2)

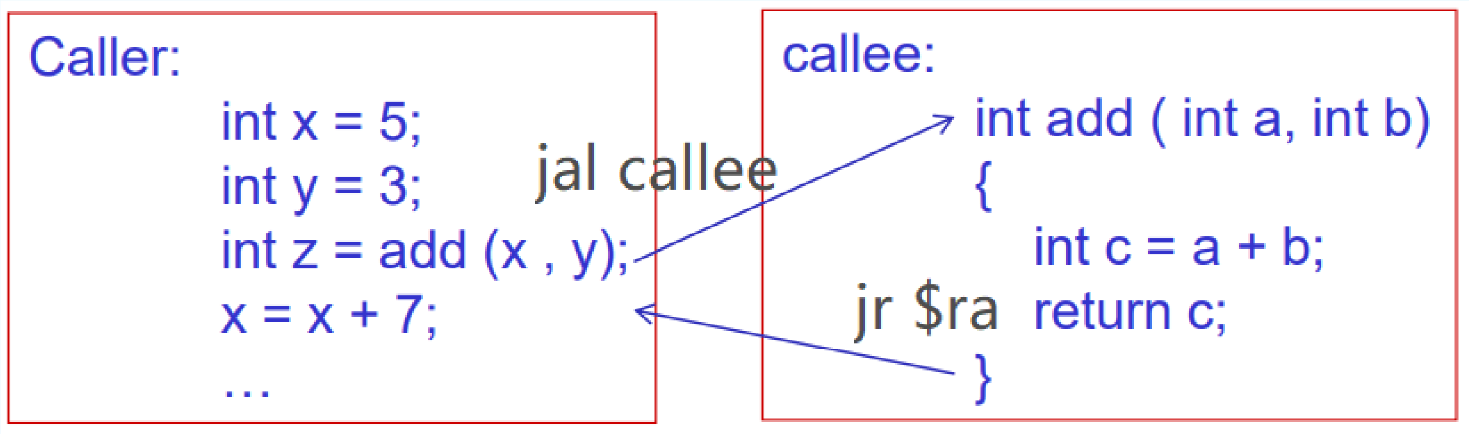
Procedure & Memory

Topics

- Procedure
  - ✓ Caller & Callee
  - ✓ Stack
  - ✓ Recursion
- Memory
  - ✓ Static Data vs Dynamic Data
  - ✓ Stack vs Heap
- Practice

Procedure: Caller & Callee(1)

- Caller: The program that instigates a procedure and provides the necessary parameter values.
- Callee: A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller.
- Return Address: A link to the calling site that allows a procedure to return to the proper address; in RISC-V it is stored in register **x1(ra)**.



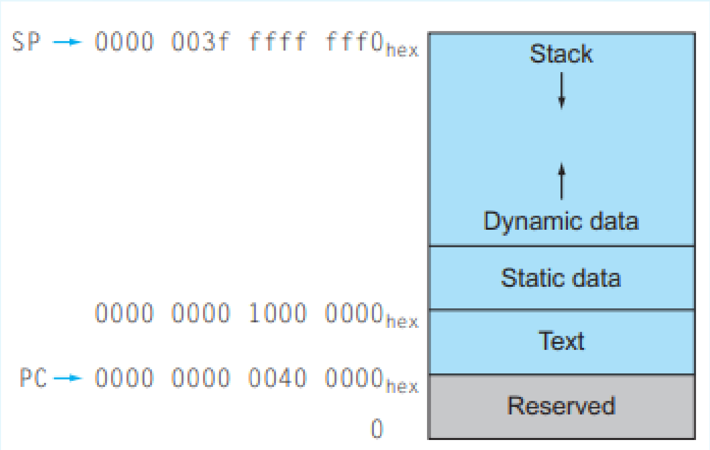
Procedure: Caller & Callee(2)

- **jal rd, function_lable** #jump-and-link instruction
  - **Save** return address (related to PC) in **register rd.**
  - **Unconditionally jump** to the instruction at function_lable.
  - Used in **caller** while calling the function.
- **jalr rd, rs1, imm** #jump-and-link register instruction
  - **Save** return address in **register rd.**
  - **Unconditionally jump** to the instruction according the sum of register rs1 and imm.
  - jalr x0, x1, 0 can be used in **callee** while returning to the caller.
- Some extended/pseudo instructions: j, jr, …
- Limit of destination address
  - For jal instruction, function_lable is a 20-bit value, adding a 0 at the end, and then be sign-extended to 32-bit. So the jumping range is **PC +/- 1MB**.
  - For jalr instruction, imm is a 12-bit value, is a 20-bit value, being sign-extended to 32-bit. So the jumping range is value of **RS1 +/- 2KB**.

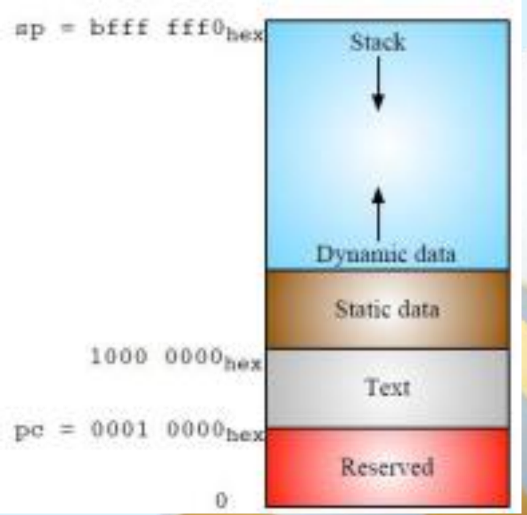
Procedure: Stack Segment

- Stack: A data structure for spilling registers organized as a last-in-first-out queue.
- Stack Pointer: A value denoting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found. In RISC-V, it is register sp, or x2.
  - ✓ In the RISC-V software specification, the stack pointer (sp) starts to grow downwards from.
  - ✓ Like dynamic data, the maximum size of a program's stack is not known in advance.
  - ✓ As the program pushes values on the stack, the operating system expands the stack segment down, toward the data segment.
  - ✓ In Rars, the memory is allocated as follows.

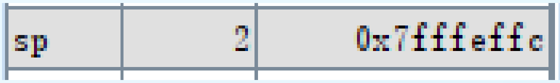
The RISC-V memory allocation for program and data (64-bit)

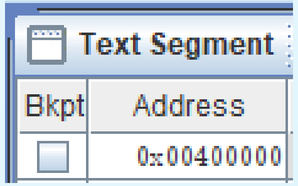


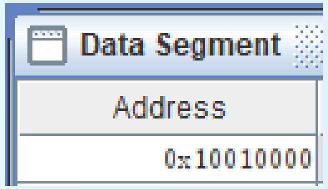
The RISC-V memory allocation for program and data (32-bit)



SP in Rars







Procedure: Stack Segment demo(1)

- Run the demo, learn the push and pop operation of stack, and answer questions.
  - Q1: What is the value of register **ra** before calling “print_string” each time?
  - Q2: Is it OK to remove the push and pop processing of ra on the stack in “print_string” ?
  - Q3: Is it OK to remove the push and pop processing of a0 on the stack in “print_string” ?

```
# Piece 4-1-1
.data
	tdata: .space 6
	str1:  .asciz "\nThe orignal string is: "
	str2:  .asciz "\nThe last two character of the string is: "
.text
	la a0, tdata
	addi a1, zero, 6
	li a7, 8
	ecall
```

```
# Piece 4-1-2
	la a0, str1
	jal print_string

	la a0, tdata
	jal print_string

	la a0, str2
	jal print_string

	la a0, tdata
	addi a0, a0, 3
	jal print_string

	li a7, 10
	ecall
```

```
# Piece 4-1-3
print_string:
	addi sp, sp, -8
	sw ra, 4(sp)
	sw a0, 0(sp)
	li a7, 4
	ecall
	lw a0, 0(sp)
	lw ra, 4(sp)
	addi sp, sp, 8
	jr ra
```

Procedure: Stack Segment demo(2)

- Run the demo, and answer questions.
  - Q1: Is it OK to remove the push and pop processing of register ra on the stack in "print_string", "print_new_line", "print_dec_result", and "print_hex_result" ?
  - Q2: What about register a0?

```
# Piece 4-2-1
.data
	tdata: .word 0x00000001
	str1:  .asciz "\nThe result is: "
	str2:  .asciz "\n"
.text
	li a7, 5
	ecall
	lw a1, tdata
	add a0, a0, a1
	mv t0, a0
```

```
# Piece 4-2-2
	la a0, str1
	jal print_string

	mv a0, t0
	jal print_dec_result
	jal print_new_line
	jal print_hex_result

	li a7, 10
	ecall
```

```
# Piece 4-2-3
print_string:
	addi sp, sp, -8
	sw ra, 4(sp)
	sw a0, 0(sp)
	li a7, 4
	ecall
	lw a0, 0(sp)
	lw ra, 4(sp)
	addi sp, sp, 8
	jr ra
```

```
# Piece 4-2-4
print_new_line:
	addi sp, sp, -8
	sw ra, 4(sp)
	sw a0, 0(sp)
	la a0, str2
	li a7, 4
	ecall
	lw a0, 0(sp)
	lw ra, 4(sp)
	addi sp, sp, 8
	jr ra
```

```
# Piece 4-2-5
print_dec_result:
	addi sp, sp, -8
	sw ra, 4(sp)
	sw a0, 0(sp)
	li a7, 1
	ecall
	lw a0, 0(sp)
	lw ra, 4(sp)
	addi sp, sp, 8
	jr ra
```

```
# Piece 4-2-6
print_hex_result:
	addi sp, sp, -8
	sw ra, 4(sp)
	sw a0, 0(sp)
	li a7, 34
	ecall
	lw a0, 0(sp)
	lw ra, 4(sp)
	addi sp, sp, 8
	jr ra
```

Procedure: Recursion

- ***"fact"*** is a function to calculate the Calculate the factorial.
- Run the demo, and answer the question: While calculate ***fact(6)***, how many times does push and pop processing on stack happen? How does the value of a0 change when calculate ***fact(6)***?

Code in C

```
int     fact(int n) {
        if(n<1)
                return 1;
        else
                return (n*fact(n-1));
}
```

Code in RISC-V

```
# Piece 4-3
.include "macro_print_str.asm"
.text
main:
        print_string("Please enter an integer: ")
        li a7, 5
        ecall                   #get n, and set in register a0
        jal fact                #call the fact function
        li a7, 1
        ecall
        end
fact:
        addi sp, sp,-8                  #adjust stack for 2 items
        sw ra, 4(sp)            #save the return address
        sw a0, 0(sp)            #save the argument n

        slti t0, a0, 1          #test for n<1
        beq t0, zero, L1        #if n>=1,go to L1

        addi a0, zero, 1        #else return 1
        addi sp, sp, 8                  #pop 2 items off stack
        jr ra                   #return to caller

L1:     addi a0, a0, -1                 #n>=1; argument gets(n-1)
        jal fact                #call fact with(n-1)

        addi t1, a0, 0                  #
        lw a0, 0(sp)            #return from jal: restore argument
        lw ra, 4(sp)            #restore the return address
        addi sp, sp, 8                  #adjust stack pointer to pop 2 items

        mul a0, a0, t1                  #return n*fact(n-1)
        jr ra                   #return to the caller
```

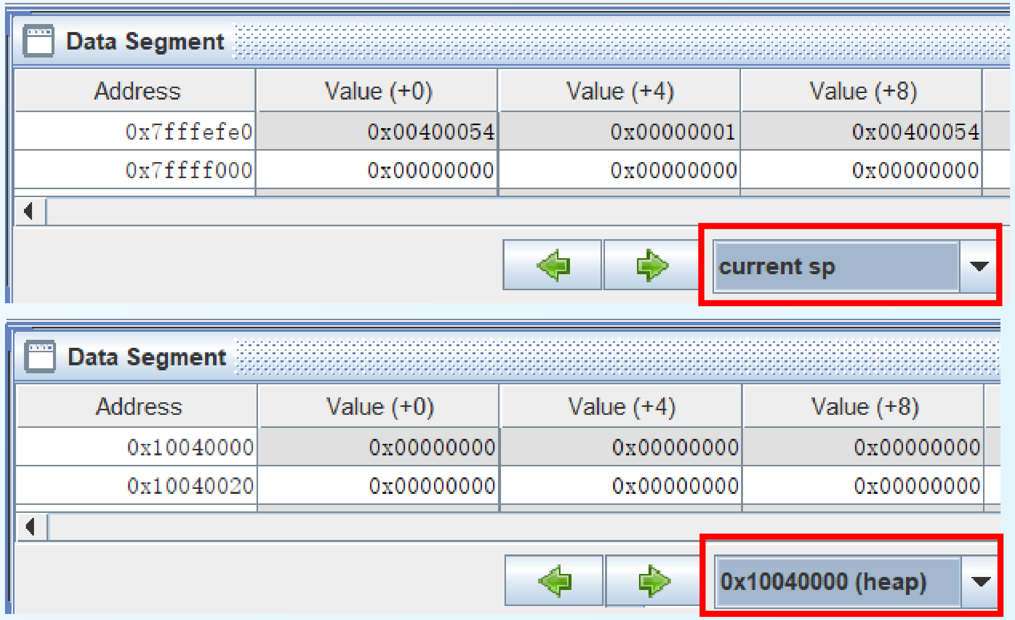
Memory: Static Data vs Dynamic Data

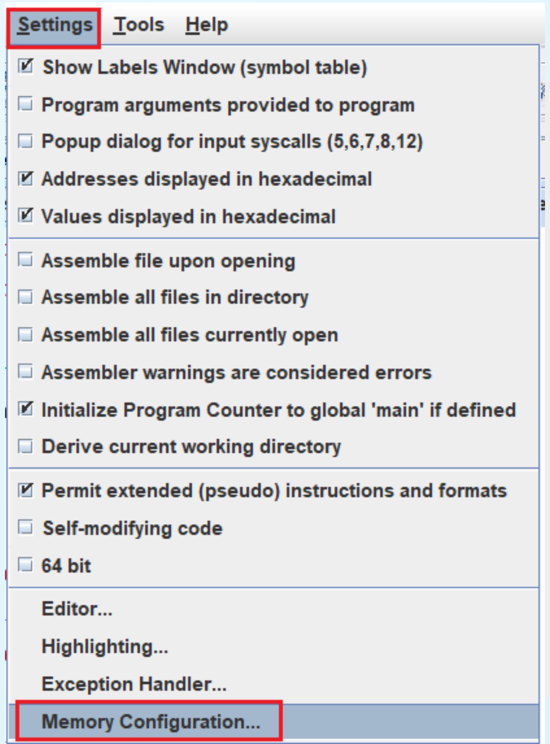
- **Static data**
  - ✓The portion of memory that contains data whose size is **known** to the compiler and whose lifetime is the program's entire execution.
  - ✓To simplify access to static data, some RISC-V compilers reserve a register **x3** for use as the global pointer, or **gp**.
  - ✓In Rars, we use ".data" to explicit static data.
- **Dynamic data**
  - ✓Allocated by malloc() in C and by new in Java.
  - ✓Including heap and stack segment.
  - ✓In Rars, we use **Sbrk**(NO. 9) system call to allocate heap memory.

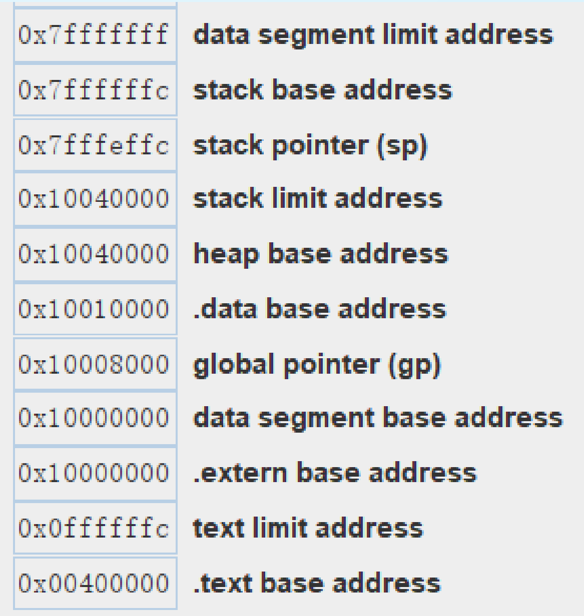
Sbrk	9	Allocate heap memory

Memory: Stack vs Heap

- **Stack**: Be used to store the local variable. It's continuous in memory.
- **Heap**: Heap is a discontinuous memory allocation method, and commonly used for dynamic allocation and release of memory.
- The stack and heap grow toward each other, thereby allowing the efficient use of memory as the two segments wax and wane.







Practice 1

- The **Fibonacci sequence**, also known as the golden ratio sequence, was introduced by mathematician Leonardo Fibonacci using rabbit breeding as an example, and is also known as the "rabbit sequence".
- Its values are: 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- Its definition: $F(0) = 1, F(1) = 1, F(n) = F(n-1) + F(n-2) \quad (n \geq 2,\ n \in N^*)$
- Please use recursive methods to complete the calculation about the Fibonacci sequence.
  - ✓ Input a integer *m*.
  - ✓ Use *m* as the index, and output the m^{th} number in Fibonacci sequence.
  - ✓ The 0^{th} number in Fibonacci sequence is 1, and the 1^{st} number is 1, the 2^{nd} number is 2, ……

Practice 2 (1)

```
# Piece 4-4-1
.include "macro_print_str.asm"
.data
	min_value: .word 0
.text
	print_string("please input the number:")
	li a7, 5		#read an integer
	ecall
	mv t0, a0	#t0 is the number of integers

	slli a0, t0, 2		#new a heap with 4*t0
	li a7, 9	#a0 is both used as argument and return value
	ecall
	mv t1, a0	#t1 is the start of the heap
	mv t2, a0	#t2 is the pointer
print_string("please input the array\n")
	add t3, zero, zero		#set t3 as i

loop_read:
	li a7, 5	#read the array
	ecall
	sw a0, (t2)

	addi t2, t2, 4
	addi t3, t3, 1
	bne t3, t0, loop_read
```

```
# Piece 4-4-2
	lw a0, (t1)	#initialize the min_value
	la t4, min_value
	sw a0, (t4)
	add t3, zero, zero
	add t2, t1, zero	#t1 is the start of the heap

loop_find_min:
	lw a0, min_value
	lw a1, (t2)
	jal find_min
	la t4, min_value
	sw a0, (t4)
	addi t2, t2, 4			#t2 is the pointer
	addi t3, t3, 1
	bne t3, t0, loop_find_min  #t0 is the number of integers

	print_string("the min value is: ")
	li a7, 1
	la t4, min_value
	lw a0, (t4)
	ecall

	end

find_min:
	blt a0, a1, not_update
	mv a0, a1

not_update:
	jr ra
```

```
please input the number:3
please input the array
-5
0
45
the min value is: -5
-- program is finished running (0) --
```

Practice 2 (2)

- The demo on previous page is supposed to get and store the data from user input, find and output the minimal value among the data. Answer the questions below.
  - Q1. What's the value of register a0 after finish executing the system call with yellow background ? Is it the same with the value of register sp?
  - Q2. On what addresses are the input data stored in memory? Are the addresses belong to static storage or dynamic storage? Are the addresses belong to stack or heap?
  - Q3. On what address is the minimal data stored in memory? Is the address belong to static storage or dynamic storage? Is the address belong to stack or heap?
  - Q4. If the 1st input number is 0 (the number of integers), what will happen? why?
  - Q5. Modify this demo to make it better when the number is 0.