Lab 07. Subroutine Call and Passing Parameters Using the Stack

Goals

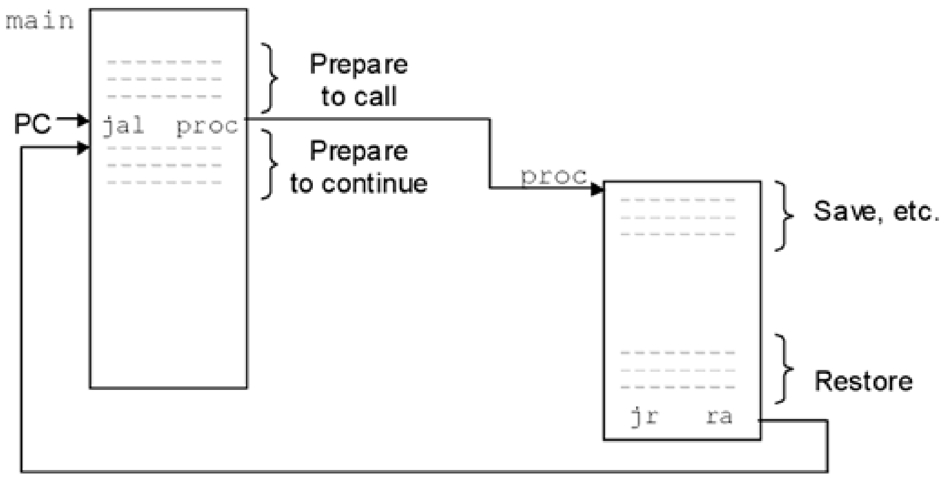
After this lab session, students will understand how to call subroutines and how the stack mechanism works. Additionally, students will be able to write their own subroutines that use the stack for passing parameters and returning results.

Subroutine Call

A procedure (subroutine) is a block of code that performs a specific task and may return one or more results based on input parameters. When the subroutine finishes, the processor returns to the location of the subroutine call to continue executing the next instructions.

In assembly programming, a subroutine is typically associated with a label to mark the starting address of the subroutine. There are two instructions used when working with subroutines:

- **`jal rd, label`** (jump and link): This saves the address of the next instruction (pc + 4) into the **rd** register and jumps to the instruction at the **label**. This address will be used to return to the main program after the subroutine finishes. The pseudo-instruction **`jal label`** (equivalent to **`jal ra, label`**) saves the return address in the **ra** register, which is commonly used to call a subroutine.
- **`jalr rd, rs1, imm`** (jump and link register): This saves the address of the next instruction (pc + 4) into the **rd** register and jumps to the instruction at address **rs1 + imm**. The pseudo-instruction **`jr ra`** (equivalent to **`jalr zero, ra, 0`**) jumps to the address stored in the **ra** register and is commonly used to return to the main program.



Relationship between the main program and a procedure.

Assignments at Home and at Lab

Home Assignment 1

The program below illustrates how to declare and use the **abs** function to calculate the absolute value of an integer. The function uses two registers: **a0** holds the input parameter and **s0** holds the result. Read the program carefully to understand how to declare and call the subroutine.

```
# Laboratory Exercise 7 Home Assignment 1
.text
main:
    li  a0, -45      # load input parameter
    jal abs          # jump and link to abs procedure

    li  a7, 10       # terminate
    ecall
end_main:
# ----------------------------------------------------------------------
# function abs
# param[in]    a0     the interger need to be gained the absolute
value
# return       s0     absolute value
# ----------------------------------------------------------------------
abs:
    sub s0, zero, a0    # put -a0 in s0; in case a0 < 0
    blt a0, zero, done  # if a0<0 then done
    add s0, a0, zero    # else put a0 in s0
done:
    jr  ra
```

Home Assignment 2

In this example, the **max** subroutine is declared and used to find the largest element among three integers. The parameters are passed to the subroutine via the **a0**, **a1**, and **a2** registers, and the result is stored in the **s0** register. Read the program carefully to understand how to declare and call the subroutine.

```
# Laboratory Exercise 7, Home Assignment 2
.text
main:
    li    a0, 2     # load test input
    li    a1, 6
    li    a2, 9
    jal   max       # call max procedure

    li    a7, 10    # terminate
    ecall
end_main:

# ----------------------------------------------------------------------
# Procedure max: find the largest of three integers
# param[in]  a0  integers
# param[in]  a1  integers
# param[in]  a2  integers
```

```
# return     s0   the largest value
# -----------------------------------------------------------------------
max:
    add     s0, a0, zero    # copy a0 in s0; largest so far
    sub     t0, a1, s0      # compute a1 - s0
    blt     t0, zero, okay  # if a1 - v0 < 0 then no change
    add     s0, a1, zero    # else a1 is largest thus far
okay:
    sub     t0, a2, s0      # compute a2 - v0
    bltz    t0, zero, done  # if a2 - v0 <0 then no change
    add     s0, a2, zero    # else a2 is largest overall
done:
    jr      ra              # return to calling program
```

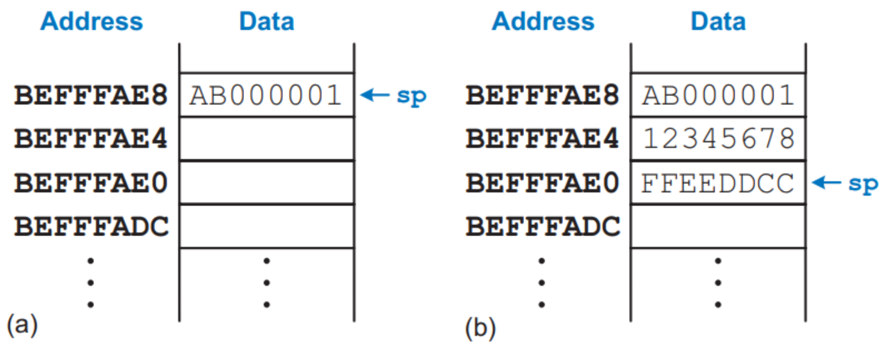
Home Assignment 3

Support:

Support: Stack memory operates arccording to the *Last In First Out* (LIFO) principle and is usually managed by the **sp** (stack pointer) register. The **sp** register stores the address of the top element of the stack and is used for two operations: **push** and **pop**.
Using the Stack:

1. Create space to store the contents of one or more registers (reduce the value of the **sp** register by the required number of bytes).
2. Store the contents of the necessary registers into the stack (using the **sw** instruction).
3. Execute the program that uses these registers.
4. Restore the original values of the registers (using the **lw** instruction).
5. Return the allocated stack memory by restoring the original value of the **sp** register.

Note that in RISC-V, the bottom of the stack has the highest address.



Stack before the push operation   Stack after the push operation

The assembly program below demonstrates how to use the stack with the **push** and **pop** operations, which are implemented by the **lw** and **sw** instructions. The values of the two registers **s0** and **s1** will be swapped using the stack.

```
# Laboratory Exercise 7, Home Assignment 3
.text
push:
    addi    sp, sp, -8      # adjust the stack pointer
    sw      s0, 4(sp)       # push s0 to stack
```

```
    sw      s1, 0(sp)       # push s1 to stack
work:
    nop
    nop
    nop
pop:
    lw      s0, 0(sp)       # pop from stack to s0
    lw      s1, 4(sp)       # pop from stack to s1
    addi    sp, sp, 8       # adjust the stack pointer
```

Support:

According to RISC-V conventions, input parameters are typically stored in the `a0-a7` registers, while return values are stored in the `a0` register. There are some questions to address:

1. What happens if a function has more than 8 input parameters or returns more than one value?
2. What happens if a function needs to save its input parameters and state to call another function?
3. What happens if a program has too many local variables to be stored in 32 registers?

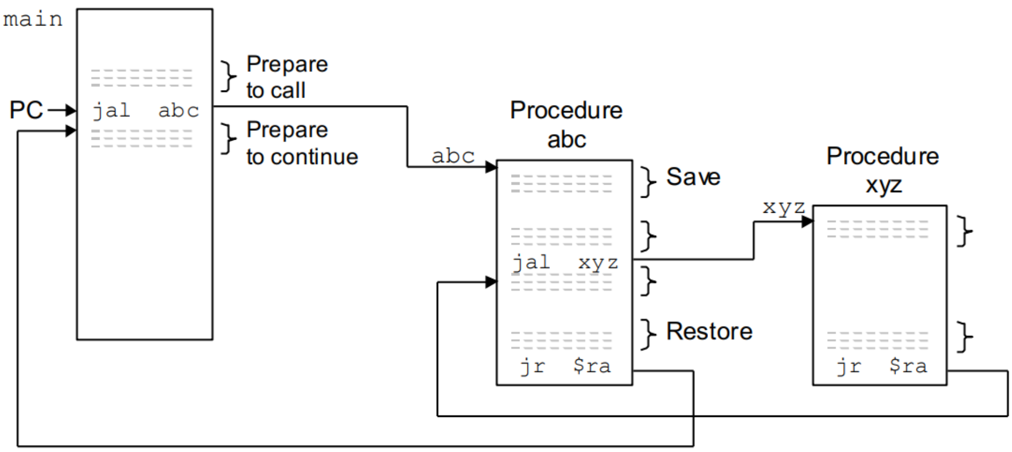
Solution: Use the stack memory.

Caller and Callee Conventions

1. **Caller Rule:** Before calling the callee, the caller saves the contents of the registers containing the input parameters and its temporary registers onto the stack (`a0-a7` and `t0-t6`), allowing the callee to use these registers. After the callee finishes, the caller restores the contents of the saved registers.
2. **Callee Rule:** Before executing, the callee must save the contents of the registers it wants to use (`s0-s11` and `ra`). Before returning to the caller, the callee must restore the contents of the previously saved registers.

These conventions ensure that the subroutine can use as many registers as possible to serve the program.

Note for Nested Procedures:



When calling the **xyz** subroutine, you must save the **`ra`** register (which currently holds the return address of the **abc** function) onto the stack before jumping to **xyz**. Otherwise, the current value of the **`ra`** register will be overwritten by the return address of the **xyz** subroutine, and the function will not be able to return to the main program.

Home Assignment 4

The following program uses a recursive algorithm to calculate **n!**. Read the program carefully to understand how the stack is used to store and restore registers.

```
# Laboratory Exercise 7, Home Assignment 4
.data
message: .asciz  "Ket qua tinh giai thua la: "

.text
main:
    jal     WARP

print:
    add     a1, s0, zero     # a0 = result from N!
    li      a7, 56
    la      a0, message
    ecall

quit:
    li      a7, 10           # terminate
    ecall
end_main:

# ---------------------------------------------------------------------
# Procedure WARP: assign value and call FACT
# ---------------------------------------------------------------------
WARP:
    addi    sp, sp, -4   # adjust stack pointer
    sw      ra, 0(sp)    # save return address

    li      a0, 3        # load test input N
    jal     FACT         # call fact procedure

    lw      ra, 0(sp)    # restore return address
    addi    sp, sp, 4    # return stack pointer
    jr      ra
wrap_end:

# ---------------------------------------------------------------------
# Procedure FACT: compute N!
# param[in]  a0  integer N
# return     s0   the largest value
# ---------------------------------------------------------------------
FACT:
    addi     sp, sp, -8  # allocate space for ra, a0 in stack
    sw       ra, 4(sp)   # save ra register
    sw       a0, 0(sp)   # save a0 register

    li       t0, 2
    bge      a0, t0, recursive
    li       s0, 1       # return the result N!=1
    j        done
```

```
recursive:
    addi    a0, a0, -1  # adjust input argument
    jal     FACT        # recursive call
    lw      s1, 0(sp)   # load a0
    mul     s0, s0, s1
done:
    lw      ra, 4(sp)   # restore ra register
    lw      a0, 0(sp)   # restore a0 register
    addi    sp,sp,8     # restore stack pointer
    jr      ra          # jump to caller
fact_end:
```

Assignment 1

Create a project to implement Home Assignment 1. Compile and simulate it. Change the program parameters (register **a0**) and observe the execution results. Run the program in the single-step mode and pay attention to the changes in registers, especially the **pc** and **ra** registers.

Assignment 2

Create a project to implement Home Assignment 2. Compile and simulate it. Change the program parameters (registers **a0**, **a1**, **a2**) and observe the execution results. Run the program in the single-step mode and pay attention to the changes in registers, especially the **pc** and **ra** registers.

Assignment 3

Create a project to implement Home Assignment 3. Compile and simulate it. Change the program parameters (registers **s0**, **s1**), observe the process and results. Pay attention to changes in the **sp** register. Observe the memory pointed to by **sp** in the Data Segment window.

Assignment 4

Create a project to implement Home Assignment 4. Compile and simulate it. Change the parameter in the **a0** register and check the result in the **s0** register. Run the program in the single-step mode and observe the changes in the registers **pc**, **ra**, **sp**, **a0**, **s0**. List the values in the stack memory when executing the program with n = 3.

Assignment 5

Write a subroutine to find the largest value, the smallest value, and their respective positions in a list of 8 integers stored in the registers from **a0** to **a7**. For example:

- Largest: 9, 3 → The largest value is 9, stored in a3.
- Smallest: -3, 6 → The smallest value is -3, stored in a6.

Hint: Use the stack memory to pass parameters.