Bài 7. Lệnh gọi chương trình con, truyền tham số sử dụng ngăn xếp

Họ và tên: Nguyễn Thành Duy

MSSV: 20235696

Assignment 1

Thanh ghi ra, pc trước khi chạy chương trình

ra	1	0x00000000
sp	2	0x7fffeffc

Sau câu lệnh đầu tiên:

pc		0x00400004

Sau khi chạy dòng lệnh:

```
jal abs # jump and link to abs procedure
```

Thanh ghi ra: Thanh ghi ra chứa địa chỉ của câu lệnh tiếp theo của chương trình

ra	1	0x00400008

Chương trình nhảy đến nhãn **abs** để thực hiện câu lệnh của nhãn:

```
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:
```

Sau câu lệnh: Chương trình nhảy tới địa chỉ ra tức là địa chỉ của dòng lệnh kế tiếp trước khi nhảy sang thực hiện chương trình ở nhãn abs.

```
jr ra
```

Assignment 2

Thanh ghi **ra**, **pc** trước khi chạy chương trình:

ra	1	0x00000000
sp	2	0x7fffeffc

pc		0x00400000

Sau hai 3 câu lệnh:

```
li a0, 2 # load test input
li a1, 6
```

```
li a2, 9
```

Thanh ghi pc:

Câu lệnh tiếp theo:

```
jal max # call max procedure
```

Sau khi chạy thấy thanh ghi **a0** chứa địa chỉ của câu lệnh tiếp theo và chương trình nhảy tới nhãn max:

Sau khi thực hiện chương trình ở nhãn 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
blt t0, zero, done # if a2 - v0 <0 then no change
add s0, a2, zero # else a2 is largest overall
done:
```

Sau khi thực hiện câu lệnh: chương trình nhảy tới đoạn chương trình tiếp theo tức là đoạn chương trình mà **ra** lưu địa chỉ trước đó.

```
jr ra # return to calling program
```

Assignment 3

Thanh ghi sp trước khi chạy chương trình:

Câu lệnh tiếp theo gán s0 = 5, s1 = 7

```
li s0, 5
li s1, 7
```

Câu lệnh: Giảm giá trị của stack pointer (sp) đi 8 bytes để dành chỗ cho 2 giá trị (4 bytes mỗi thanh ghi)

```
addi sp, sp, -8 # adjust the stack pointer
```

Kết quả:

sp	2	0x7fffeff4

Câu lệnh tiếp theo: đẩy s0, s1 vào stack

```
sw s0, 4(sp) # push s0 to stack
```

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7fffefe0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000005	0x00000000

```
sw s1, 0(sp) # push s1 to stack
```

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7fffefe0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000007	0x00000005	0x00000000

Hiện tại đỉnh ngăn xếp sp đang lưu giá trị s1 = 7.

Câu lệnh tiếp theo là lấy giá trị trỏ bởi sp ra ngăn xếp và lưu vào s0

```
lw s0, 0(sp) # pop from stack to s0
```

s0	8	0x00000007

Câu lệnh tiếp theo là lấy giá trị trỏ bởi sp ra ngăn xếp và lưu vào s1

```
lw s1, 4(sp) # pop from stack to s1
```

s1	9	0x00000005

Sau khi lấy dữ liệu, con trỏ sp được tăng lên 8 để quay lại trạng thái trước khi push.

```
addi sp, sp, 8 # adjust the stack pointer
```

sp	2	0x7fffeffc

Assignment 4

Thanh ghi ra, pc, sp, a0, s0 trước khi chạy chương trình:

Registers | Floating Point | Control and Status

Name	Number	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x7fffeffc
gp	3	0x10008000
tp	4	0x00000000
t0	5	0x00000000
t1	6	0x00000000
t2	7	0x00000000
s0	8	0x00000000
s1	9	0x00000000
a0	10	0x00000000
a1	11	0x00000000
a2	12	0x00000000
a3	13	0x00000000
a4	14	0x00000000
a5	15	0x00000000
a6	16	0x00000000
a7	17	0x00000000
s2	18	0x00000000
s3	19	0x00000000
s4	20	0x00000000
s5	21	0x00000000
s6	22	0x00000000
s7	23	0x00000000
s8	24	0x00000000
s9	25	0x00000000
s10	26	0x00000000
s11	27	0x00000000
t3	28	0x00000000
t4	29	0x00000000
t5	30	0x00000000
t6	31	0x00000000
pc		0x00400000

Sau câu lệnh: Thanh ghi ra được gán giá trị địa chỉ của câu lệnh tiếp theo

```
jal WARP
```

ra	1	0x00400004

Câu lệnh sau làm thay đổi thanh ghi sp, giá trị thanh ghi trừ 4 bytes

```
addi sp, sp, -4 # adjust stack pointer
```

sp	2	0x7fffeff8

Câu lệnh sau để đẩy địa chỉ ra (địa chỉ của câu lệnh tiếp theo) vào stack

```
sw ra, 0(sp) # save return address
```

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7fffefe0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00400004	0x00000000

Địa chỉ của thanh ghi pc, ra trước và sau khi chạy câu lệnh:

```
jal FACT # call fact procedure
```

Trước:

ra	1	0x00400004
sp	2	0x7fffeff8
pc		0x0040002c

Sau:

ra	1	0x00400030
sp	2	0x7fffeff8
pc		0x0040003c

Thanh ra lưu địa chỉ của câu lệnh tiếp theo ngay phía sau dòng lệnh trên

Câu lệnh sau có chức năng lưu giá trị và địa chỉ của dòng lệnh tiếp theo vào ra, a0 và khôi phục lại địa chỉ sp trước đó.

```
lw ra, 4(sp) # restore ra register
lw a0, 0(sp) # restore a0 register
addi sp,sp,8 # restore stack pointer
```

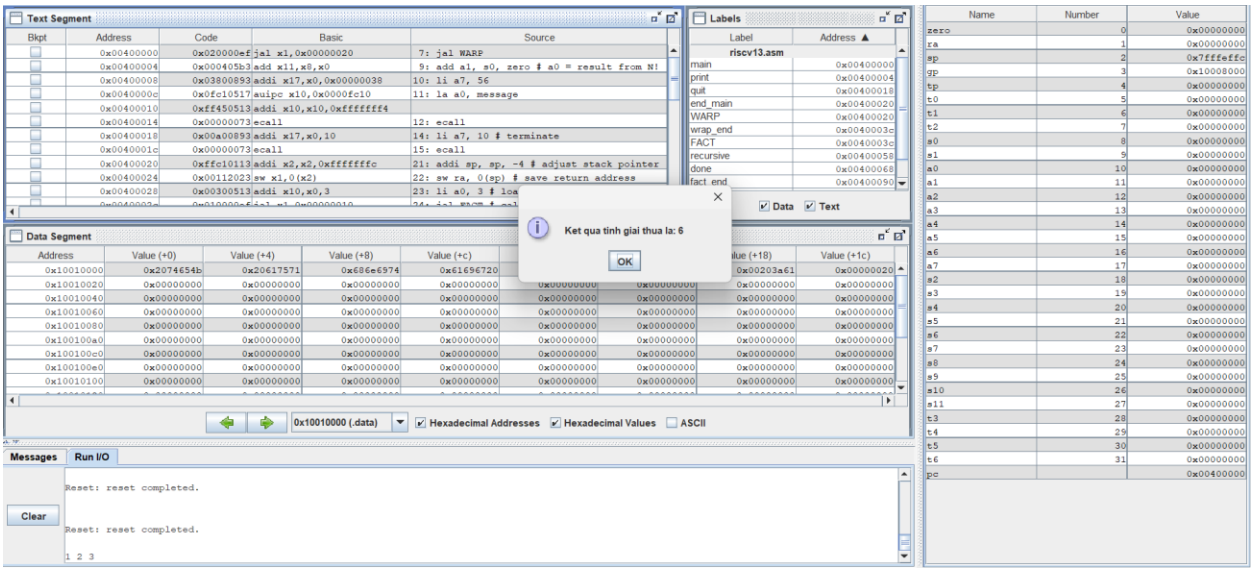
Sau khi load được địa chỉ của câu lệnh tiếp theo trước đo vào ra thì chương trình sẽ nhảy tới câu lệnh tiếp theo đó.

```
jr ra # jump to caller
```

Câu lệnh sau đây dùng để tính giai thừa và lặp lại cho tới khi s0 = 1

```
recursive:
addi a0, a0, -1 # adjust input argument
jal FACT # recursive call
lw s1, 0(sp) # load a0
mul s0, s0, s1
```

- Các giá trị trong vùng nhớ ngăn xếp khi thực hiện chương trình với n = 3



Assignment 5

Code:

```
.data
largest_msg: .asciz "Largest: "
smallest_msg: .asciz "Smallest: "
comma: .asciz ", "
newline: .asciz "\n"

.text

main:
   # Khởi tạo giá trị cho các thanh a0-a7
   li a0, 5
   li a1, -5
   li a2, 7
   li a3, 10
   li a4, 2
   li a5, 0
   li a6, -1
   li a7, 4
   # Lưu các giá trị vào stack
   addi sp, sp, -32
   sw a0, 0(sp)
   sw a1, 4(sp)
   sw a2, 8(sp)
   sw a3, 12(sp)
   sw a4, 16(sp)
```

```
sw a5, 20(sp)
sw a6, 24(sp)
sw a7, 28(sp)
li s10, 8
# Gọi hàm tìm min/max
mv a0, sp  # Truyền địa chỉ mảng
jal ra, find_min_max

# In kết quả
li a7, 4
la a0, largest_msg
ecall

li a7, 1
mv a0, t0  # In giá trị lớn nhất
ecall

li a7, 4
la a0, comma
ecall

li a7, 1
mv a0, t1  # In vị trí lớn nhất
ecall

li a7, 4
la a0, newline
ecall

li a7, 4
la a0, smallest_msg
ecall

li a7, 1
mv a0, t2  # In giá trị nhỏ nhất
ecall

li a7, 4
la a0, comma
ecall

li a7, 1
mv a0, t3  # In vị trí nhỏ nhất
ecall
```

```
    li a7, 4
    la a0, newline
    ecall

    # Kết thúc chương trình
    li a7, 10
    ecall

find_min_max:
    mv t6, a0   # Sao lưu địa chỉ gốc của stack
    lw t0, 0(t6)  # Giá trị lớn nhất ban đầu
    lw t2, 0(t6)  # Giá trị nhỏ nhất ban đầu
    li t1, 0      # Vị trí lớn nhất ban đầu
    li t3, 0      # Vị trí nhỏ nhất ban đầu

    li t4, 1  # Vòng lặp bắt đầu từ phần tử thứ 2
    addi t6, t6, 4  # Trỏ đến phần tử thứ hai

loop:
    bge t4, s10, end_loop  # Nếu đã xét hết 8 phần tử, thoát vòng lặp

    lw t5, 0(t6)  # Lấy giá trị hiện tại từ stack

    # So sánh giá trị lớn nhất
    bgt t5, t0, update_max
    j check_min

update_max:
    mv t0, t5
    mv t1, t4

check_min:
    blt t5, t2, update_min
    j next

update_min:
    mv t2, t5
    mv t3, t4

next:
    addi t6, t6, 4  # Trỏ đến phần tử tiếp theo
    addi t4, t4, 1  # Tăng biến đếm
    j loop
```

```
end_loop:
  jr ra
```

Giải thích:

- Đầu vào là mảng: 5, -5, 7, 10, 2, 0, - 1, 4 được lưu tương ứng trong các thanh a0-a7
- Giá trị lớn nhất: 10 nằm ở a3
- Giá trị nhỏ nhất: -5 nằm ở a1

Kết quả mong muốn khi chạy chương trình:

Largest: 10, 3

Smallest: -5,1

Kết quả:

Như mong muốn là: Largest: 10, 3

Smallest: -5,1